Participants'
Proceedings
of the
Third International
Conference

ALGEBRAIC METHODOLOGY AND SOFTWARE TECHNOLOGY

June 21 - 25, 1993

N00014-93-1-0424

University of Twente
Enschede, The Netherlands

INRIA

dit
UPM

Mathematics well-applied illuminates rather than confuses

Participants' Proceedings

AMAST'93

Third International Conference
on
Algebraic Methodology and Software Technology

University of Twente

The Netherlands

June 21–25, 1993

Organising Committee:

General chairman:	Maurice Nivat	University of Paris VII, Paris, F
Programme chairman:	Giuseppe Scollo	University of Twente, Enschede, NL
Publicity chairman:	Charles Rattray	University of Stirling, GB
	Teodor Rus	University of Iowa, Iowa City, USA
	V.S. Alagar	Concordia University, Montreal, CDN
Education Day chairman:	Hans-Jörg Kreowski	University of Bremen, D
Local chairman:	Ed Brinksma	University of Twente, Enschede, NL
Finance chairman:	Frans van der Avert	University of Twente, Enschede, NL
Members:	Mohammed Bettaz	University of Constantine, DZ
	Christine Choppy	University of Paris-Sud, LRI, Orsay, F
	Pierre Deransart	INRIA, Rocquencourt, F
	Arthur Fleck	University of Iowa, Iowa City, USA
	Luigi Logrippo	University of Ottawa, Ontario, CDN
	Michael O'Donnell	University of Chicago, USA
	Juan Quemada	University of Madrid, E
	Ralph Wachter	Office of Naval Research, Arlington, USA

Local Committee: Herman Balsters, Han Bäumer, Pim van den Broek, Rolf de By,
Maarten Fokkinga, Pim Kars, Mark van de Voort, Job Zwiers

Secretariat: Joke Lammerink, Alice Hoogvliet-Haverkate, Charlotte Bijron

Programme Committee:

Martin Abadi	DEC, Palo Alto, USA	Michael Mislove	Tulane U., New Orleans, USA
Egidio Astesiano	U. Genova, I	Ugo Montanari	U. Pisa, I
C.-B. Ben-Yelles	USTHB, Alger, DZ	Peter D. Mosses	Aarhus U., DK
Michel Bidoit	DMI/LIENS, Paris, F	Istvan Németi	Acad. Sci., Budapest, H
Wim Blok	U. Illinois, Chicago, USA	*Don Pigozzi*	*Iowa State U., Ames, USA*
Chris Brink	U. Cape Town, ZA	Don Sannella	U. Edinburgh, GB
Pierre-Luis Curien	DMI/LIENS, Paris, F	R.K. Shyamasundar	TIFR, Bombay, IND
Kokichi Futatsugi	EL/CLS, Tokyo, J	John Staples	U. Queensland, Brisbane, AUS
Steven Givant	Mills College, Oakland, USA	Andrzej Tarlecki	Acad. Sci., Warsaw, PL
William S. Hatcher	U. Laval, Quebec City, CDN	Frits Vaandrager	CWI, Amsterdam, NL
Douglas J. Howe	AT&T, Murray Hill, USA	Paulo A.S. Veloso	PUC, Rio de Janeiro, BR
Bjarni Jónsson	Vanderbilt U., Nashville, USA	Eric Wagner	IBM, Yorktown Heights, USA
Giancarlo Mauri	U. Milano, I	Robert F.C. Walters	U. Sydney, AUS
Ali Mili	U. Ottawa, CDN	Martin Wirsing	U. München, D

Invited Speakers:

Hajnal Andreka and Ildiko Sain	Math. Inst., Acad. Sci., Budapest, H
Hubert Comon	LRI, Université de Paris Sud, Orsay, F
Rob van Glabbeek	Stanford University, USA
Nicolas Halbwachs	INPG, Grenoble, F
Michael Johnson	Macquarie University, Sydney, AUS
Hans-Jörg Kreowski	University of Bremen, D
Giorgio Levi	University of Pisa, I
Roger D. Maddux	Iowa State University, Ames, USA
David L. Parnas	McMaster University, Ontario, CDN
Jacques Printz	Cons. Nat. des Arts et Metiers, Paris, F
Steve Schneider	PRG, Oxford University, GB

Foreword

The first two AMAST conferences, respectively held in May 1989 and May 1991 at the University of Iowa, were well received and encouraged the regular organization of further AMAST conferences on a biennial schedule.

The goal of these conferences is to foster algebraic methodology as a foundation for software technology, and to show that this can lead to practical mathematical alternatives to the ad-hoc approaches commonly used in software engineering and development.

While the AMAST goal is mainly research-oriented, the relevance of adequate mathematical education of software developers is recognized as well. In order to be effective in this direction, the first day of the third AMAST conference is dedicated to the aforementioned special interest topic. A summary of the opening talk by Hans-Jörg Kreowski and preliminary versions of the two invited papers, respectively by David L. Parnas and by Jacques Printz, are included in this proceedings. Yuri Gurevich and Istvan Németi are in charge of animating and moderating the discussion on education.

As to the research-oriented contents of the proceedings, these consist of 8 invited papers and 32 extended abstracts of selected communications. The selection was very severe, for a record of 121 submissions were received; besides the selected communications, 14 other submissions were judged to deserve presentation, but could not be selected because of the programme constraints.

The AMAST goal motivates the interest in showcasing software systems that are developed, or help development, by algebraic methods, techniques and tools. The AMAST'93 programme features seven demonstrations of such systems. Short descriptions of these systems form the closing part of this proceedings.

While the geographical scope of AMAST has rapidly grown to encompass all continents, as one can see from the contents of this proceedings, the fourth AMAST conference is expected to be held at Concordia University, Montréal, in June 1995.

The financial and organizational support by the AMAST'93 sponsors is gladly acknowledged.

We would like to thank Ms. Charlotte Bijron, Ms. Alice Hoogvliet-Haverkate, Ms. Joke Lammerink, and Ms. Yvonne Rokker for their excellent taking care of the conference secretariat.

Finally, we owe special thanks to Yuri Gurevich for allowing us to open the proceedings with the text of the intriguing banquet speech which he delivered to the second AMAST meeting. Starting with his humour and *finesse d'esprit* will certainly set the third AMAST meeting in the best mood towards the accomplishment of its goals.

AMAST'93 Organizing Committee
Enschede, June 1993

AMAST'93 Sponsors

The AMAST'93 conference is made possible by the financial and organizational support of the following institutions:

- Commission of the European Communities, within the ESPRIT Basic Research Programme
- Office of Naval Research
- University of Twente
- University of Iowa
- University of Stirling
- Institut National de Recherche en Informatique et Automatique (INRIA)
- University of Paris VII, LITP
- Concordia University, Montréal
- University of Ottawa
- University of Constantine
- University of Madrid
- University of Chicago

The AMAST'93 conference is held under the auspices and with the cooperation of the following associations:

- European Association for Theoretical Computer Science (EATCS)
- Association for Symbolic Logic (ASL)
- British Computer Society/Formal Aspects of Computing Science (BCS/FACS)
- ESPRIT Basic Research Working Groups COMPASS and ASMICS

Cooperation is pending with the following associations:

- Association for Computing Machinery (ACM), SIGACT and SIGSOFT
- IEEE Computer Society

Advance Programme

Third International Conference on Algebraic Methodology and Software Technology, AMAST'93

University of Twente, The Netherlands, June 21–25, 1993

The goal of the third AMAST conference to be held on June 21–25, 1993, at the University of Twente, Enschede, The Netherlands, is to consolidate the trend towards using algebraic methodology as a foundation for software technology, and to show that universal algebra provides a practical mathematical alternative to the common, ad-hoc approaches to software engineering and development. Academia and industry are both beneficiaries of such a formal foundation.

To achieve the goal of the conference we aim to provide a forum in which leading researchers in mathematics, computer science, and software development, will come together to identify algebraic methodologies that are applicable as viable alternatives to the present software development approaches and to discuss the appropriateness of such alternatives with a view to implementation.

Education Day (Monday 21 June)

While the AMAST goal is mainly research-oriented, the relevance of education is recognized as well. In particular, the adequacy of the mathematical education of designers, implementors, users and maintainers of software artifacts, is recognized as being of special interest. The evaluation of, and the provision of recommendations about, the mathematical training of software developers is a necessary means to achieve that adequacy. In order to be effective in this direction, the first day of the conference will be dedicated to this special interest topic. This Education Day will start with an introductory talk by the Education Day Chairman, setting general objectives and guidelines, and proceed with two sessions; each session will have an Invited Speaker, talking about *mathematical education of the software engineer*, and a Session Moderator, controlling and animating the subsequent open discussion.

As a preliminary indication, the Education Day should aim at answering such qrestions as:

- How do we educate software designers, implementors, users, maintainers?
- What should be the ideal mathematical background of a software designer, implementor, etc.?
- What do we need to add to the the conventional way of teaching mathematics to make it more acceptable, convenient, and useful to the software designer?

The programme of the first day of the conference is thus as follows:

08:30–09:30	Registration	
09:30–09:45	AMAST'93 opening address	
Education Day	Opening:	
09:45–10:30	INVITED TALK: Hans-Jörg Kreowski	Univ. of Bremen, D
	Some tentative thoughts on teaching computer science	
10:30–11:00	Coffee break	
Morning Session	MODERATOR: Yuri Gurevich	Univ. of Michigan, Ann Arbor, USA
11:00–11:45	INVITED TALK: David Lorge Parnas	McMaster Univ., Ontario, CDN
	Mathematics of computation for (software and other) engineers	
11:45–13:00	Discussion	
13:00–14:30	Lunch break	
Afternoon Session	MODERATOR: Istvan Németi	Math. Inst., Acad. Sci., Budapest, H
14:30–15:15	INVITED TALK: Jacques Printz	Cons. Nat. des Arts et Metiers, Paris, F
	Mathematical training for the software developpers: a practical experience	
15:15–16:30	Discussion	
16:30–17:00	Conclusions	
17:00–	AMAST'93 welcoming reception	

Tuesday 22 June

morning

09:00–09:50 INVITED TALK:
Hajnal Andreka, Istvan Németi & Ildikó Sain (Math. Inst., Acad. Sci., Budapest):
Applying algebraic logic to logic

09:50–10:10 Coffee break

SESSION: *Algebraic metamathematics* (Chair: William S. Hatcher)

10:10–10:40 D. Pigozzi, A. Salibra (Iowa SU, U. Bari):
Dimension-complemented lambda-abstraction algebras

10:40–11:10 T. Mossakowski (U. Bremen):
Parametrized recursion theory – A tool for the systematic classification of specification methods

11:10–11:30 Coffee break

SESSION: *Extending functional languages* (Chair: Chris Brink)

11:30–12:00 T. Sheard (Oregon GIST):
Adding algebraic methods to traditional functional languages by using reflection

12:00–12:30 D. Bolignano, M. Debabi (Bull France):
A coherent type inference system for a concurrent, functional and imperative programming language

12:30–14:00 Lunch break

afternoon

14:00–14:50 INVITED TALK:
Roger D. Maddux (Iowa State Univ., Dept. Math.):
Relation algebras for reasoning about time, space, and programs

14:50–15:10 Coffee break

SESSION: *Relation algebra* (Chair: Don Pigozzi)

15:10–15:40 C. Brink, K. Britz, R.A. Schmidt (U. Cape Town, MPI Saarbrücken):
Peirce algebras

15:40–16:10 R. Berghammer, A. Haeberer, G. Schmidt, P. Veloso (UB Neubiberg, PUC Rio de Janeiro):
Comparing two different approaches to products in abstract relation algebras

16:10–16:30 Tea break

SESSION: *Order-sorted algebra* (Chair: Giancarlo Mauri)

16:30–17:00 M. Erwig (FU Hagen):
Specifying type systems with multi-level order-sorted algebra

17:00–17:30 P. Thiemann (U. Tübingen):
An overview of the SODA system

evening

19:30–20:30 System demonstrations

21:00–22:30 Concert (classic)

Wednesday 23 June

morning

09:00–09:50 INVITED TALK:
Michael Johnson and C.N.G. Dampney (Macquarie Univ., Sydney):
Category theory and information systems engineering

09:50–10:10 Coffee break

SESSION: *Category theory in software engineering* (Chair: Andrzej Tarlecki)

10:10–10:40 G. Hill (Imperial College, London):
Category theory for the configuration of complex systems

10:40–11:10 M. Cerioli, G. Reggio (U. Genova):
Algebraic-oriented institutions

11:10–11:30 Coffee break

SESSION: *Modular system design* (Chair: Egidio Astesiano)

11:30–12:00 M. Navarro, F. Orejas, A. Sanchez (UPV San Sebastian, UPC Barcelona):
On the correctness of modular systems

12:00–12:30 H. Ehrig, F. Parisi-Presicce (TU Berlin, U. L'Aquila):
Interaction between algebraic specification grammars and modular system design

12:30–14:00 Lunch break

afternoon

14:00–14:50 INVITED TALK:
Steve Schneider (Oxford Univ., PRG):
Rigorous specification of real-time systems

14:50–15:10 Coffee break

SESSION: *Real-time system specification* (Chair: Arthur Fleck)

15:10–15:40 R.K. Shyamasundar (TIFR Bombay):
Specification of hybrid systems in CRP

15:40–16:10 A. Cornell, J. Knaack, A. Nangia, T. Rus (BYU Utah, U. Iowa):
Real-time program synthesis from specifications

16:10–16:30 Tea break

SESSION: *Testing theory and applications* (Chair: Christine Choppy)

16:30–17:00 E. Brinksma (U. Twente):
On the coverage of partial validations

17:00–17:30 K. Drira, P. Azema (LAAS Toulouse):
Verifying communication protocols via testing-projection

evening

19:30–20:30 System demonstrations

21:00– Surprise event

Thursday 24 June

morning

09:00–09:50 INVITED TALK:
Rob J. van Glabbeek (Stanford Univ., Dept. CS):
Full abstraction and expressiveness in structural operational semantics

09:50–10:10 Coffee break

SESSION: *Algebraic semantics of concurrency* (Chair: Irene Guessarian)

10:10–10:40 P. Malacaria (LIENS Paris):
Equivalences of transition systems in an algebraic framework

10:40–11:10 E. Battiston, V. Crespi, F. De Cindio, G. Mauri (U. Milano):
Semantics frameworks for a class of modular algebraic nets

11:10–11:30 Coffee break

SESSION: *Process algebras* (Chair: Martin Wirsing)

11:30–12:00 D. de Frutos-Escrig (UC Madrid):
A characterization of LOTOS representable networks of parallel processes

12:00–12:30 R. Gorrieri, M. Roccetti (U. Bologna):
Towards performance evaluation in process algebras

12:30–14:00 Lunch break

afternoon

14:00–14:50 INVITED TALK:
Nicolas Halbwachs, Fabienne Lagnier, Pascal Raymond (INPG Grenoble, Verimag Lab.):
Synchronous observers and the verification of reactive systems

14:50–15:10 Coffee break

SESSION: *Modal logics and reactive systems* (Chair: Robert F.C. Walters)

15:10–15:40 F. Laroussinie, S. Pinchinat, Ph. Schnoebelen (LIFIA-IMAG Grenoble):
Translation results for modal logics of reactive systems

15:40–16:10 I.N. Kaufman, S.L. Meira (UFPE Recife):
Modal action logic in a practical specification language

16:10–16:30 Tea break

SESSION: *Design and refinement principles* (Chair: Peter D. Mosses)

16:30–17:00 A. Mokkedem, D. Mery (CRIN Nancy):
On using a composition principle to design parallel programs

17:00–17:30 N. Sabadini, S. Vigna, R.F.C. Walters (U. Milano, U. Sydney):
A notion of refinement for automata

evening

17:30–18:30 System demonstrations

19:00–23:00 Conference dinner

Friday 25 June

morning

09:00–09:50 INVITED TALK:
Hubert Comon (Univ. Paris Sud, LRI, Orsay):
Constraints in term algebras

09:50–10:10 Coffee break

SESSION: *Object-oriented design and programming, I* (Chair: Mohammed Bettaz)

10:10–10:40 E.G. Wagner (IBM Yorktown Heights):
The role of memory in object-based and object-oriented languages

10:40–11:10 R. Breu, M. Breu (TU München, Siemens Nixdorf München):
Abstract and concrete objects – An algebraic design method for object-based systems

11:10–11:30 Coffee break

SESSION: *Object-oriented design and programming, II* (Chair: Eric G. Wagner)

11:30–12:00 X.-M. Lu, T.S. Dillon (La Trobe U. Australia):
Towards an algebraic theory of inheritance in object oriented programming

12:00–12:30 M. Gogolla, I. Claßen (TU Braunschweig, TU Berlin):
An object-oriented design for the ACT ONE environment

12:30–14:00 Lunch break

afternoon

14:00–14:50 INVITED TALK:
Roberto Giacobazzi and Giorgio Levi (Univ. Pisa, Dept. CS) and Saumya K. Debray (Univ. Arizona, Dept. CS):
Joining abstract and concrete computations in constraint logic programming

14:50–15:10 Coffee break

SESSION: *Equational and logic programming* (Chair: Michel Bidoit)

15:10–15:40 J.G. Martin, J.J. Moreno-Navarro (UP Madrid):
A formal definition of an abstract Prolog compiler

15:40–16:10 V. Antimirov, A. Degtyarev (Copenhagen U. (DIKU), Kiev U.):
Completeness of equational definitions over predefined algebras

16:10–16:30 Tea break

SESSION: *Algebraic specification in software engineering* (Chair: R. K. Shyamasundar)

16:30–17:00 G.J. Loegel, C.V. Ravishankar (U. Michigan):
An algebraic approach to modeling in object-oriented software engineering

17:00–17:30 E.A. Scott (U. Surrey):
An automated proof of the correctness of a compiling specification

17:30– Closing

evening

22:00– live music in all pubs in Enschede.

AMAST'91 Banquet Talk

Yuri Gurevich

May 1991, Iowa City

Prologue

Tuesday, May 7, 1991. I sign the last grade sheet and smile at the spring sun. Finally the semester is over. A message from Teo Rus arrives. "The second conference on Algebraic Methodology and Software Technology needs a banquet speaker", writes Teo. I am very flattered. And scared. I recall a recent banquet talk in Ann Arbor. The man went on and on. I left before he finished. On the other hand, the invitation is a challenge and an opportunity. You know, sometimes we feel like philosophers if only anybody would listen. I accept the invitation before the scare gets a hold of me.

I leave my office and meet Kevin Compton, another member of the small computer theory group in our huge Department of Electrical Engineering and Computer Science. "How are you?" asks Kevin. "Well, I was fine only a few minutes ago", and I tell him about the invitation to give a banquet talk. "I do not envy you", says Kevin. Soon a message from him tells me about 5 books on public speaking in the library. I thumb the books. They have witty things on almost any subject, but do not mention algebra or software, let alone algebraic methodology and software technology. The volumes of humor are depressing. This is not it. Teo could find a professional joker to entertain the conference. At that time in Iowa it could be a national politician.

After thinking it over, I decide to take a scientific approach and write a scholarly paper. You know, another paper never hurts your vita. The scientific approach explains the use of "we" in the sequel.

The AMAST Phenomenon

The organizing principles are given by the following observation attributed to Don Knuth: The two most important questions about AI are: What is A and what is I?

What is the question complexity of AMAST? There are 5 letters in the word, but A appears twice. A closer examination reveals that there are only 3 questions:

(1) What is algebraic methodology?
(2) What is software technology?
(3) What does AND mean in the AMAST context?

The third question is the toughest of the three.

Algebraic methodology

According to Webster, methodology is "a system of methods, as in any science". Thus, algebraic methodology is a system of methods employed in algebra. Makes sense.

You may wonder how algebraic methodology is different from algebra. In algebra you search for definitions to formalize your theorems; in algebraic methodology you search for theorems to justify your definitions. It is clear that "algebraic methodology" sounds better on a grant proposal; it implies also some connection to applications.

Some folks ridicule the division of algebra or anything else into pure and applied. "Consider painting", they say, "if your paintings are bought by museums then you are a pure artist, and if your paintings are sold in a supermarket then you are an applied artist. But what if you intended to sell your paintings in a supermarket and a museum bought them? Are you a pure or applied artist?" We say: where do you find those clever folks? They all are in departments like Pure Mathematics or Physics. The distinction between pure and applied science is very important. How would DARPA know whom to support?

What is algebra?

It is clear that algebra is the essence of algebraic methodology. So let us examine what algebra is. Etymology often is a key to the meaning. We asked a few of our learned colleagues about the etymology of "algebra" and then consulted Webster. It turns out that folklore and Webster disagree on the etymology of "algebra".

Folklore: "algebra" as well as "algorithm" come from the name Al-Khowarazmi of a 9th century mathematician.

Webster: "algebra" comes from Arabic "al-jabr" which means the reunion of broken parts.

The folklore explanation would be more useful to us because it connects AM with ST in a very natural way. Nevertheless, being committed to a scholarly approach, we adapt Webster's explanation as more scientific and will try to find a good use for it as well.

Is algebra a part of mathematics?

Yes and no.

Why yes? This is obvious and well documented; see [Jane Doe], [Robert Roe], [John Smith].

Why no? We give 2 proofs: By contradiction and by authority. These proofs are specially designed to work on banquets, after a good meal with plenty of wine and before the dessert.

The proof by contradiction. If algebra is a part of mathematics then mathematics is broken into parts. The reunion of broken parts is algebra. Thus algebra = mathematics, which is not true.

The proof by authority. The famous Communist prophet Vladimir Ilich Lenin spoke about the algebra of social revolution. This places algebra into a different college, let alone a different department.

Is "yes and no" a legitimate answer? Sure. Since "AM and ST" is a legitimate title, "yes and no" is a legitimate answer. The question of what "yes and no" means will be discussed later on when we come to the second A of AMAST.

Algebra and logic

Logic methodology has been used in AMAST talks as much as algebraic methodology. This is not surprising. Algebra and logic are like Michigan and Ohio. Do you know that there was a war between Michigan and Ohio? It was about Toledo. You may think that each side wanted the other one to have Toledo, but this is not true. Each side wanted Toledo for itself. The federal government intervened and gave Toledo to Ohio. This explains the famous Michigan slogan OH-HOW-I-HATE-OHIO-STATE. Further, the federal government gave a portion of Wisconsin to Michigan. This is how Michigan became topologically disconnected. The reaction of Wisconsin is not documented.

The Toledo of algebra and logic is called "universal algebra" in algebra and "model theory" in logic. Maybe, Iran/Iraq is a better analogy because each side has its own name for the disputed part: Persian Gulf vs. Arabian Sea.

In any case, algebra and logic have a large intersection as witnessed by numerous AMAST talks. However we have

Theorem 1 *Algebra ≠ Logic.*

Proof The proof is by contradiction and related to the Russian journal "Algebra and Logic". It would be silly to have a journal "Algebra and Algebra", and the Russian Academy would not approve such a thing. □

Logics

There are many logics in the literature. Female logic, male logic, email logic, dialectical logic, mathematical logic, etc.

Male logic is all too known to be discussed here.

Email logic is all too painful to be discussed here.

Female logic is all too dangerous to be discussed here. The field of AMAST is dangerous as it is. As a matter of fact, we are going to discuss the dangers of the field. But there are prudent limits to risks taken.

Dialectical logic is sort of an art of being logical and illogical at the same time. In the SU (which means Soviet Union and is quite different from US; concatenation is not commutative), logic was divided into dialectical and formal. The first was always supported, the second was forbidden for years. Why? This is a wrong question, it is a question from a wrong logical system. A Soviet dissident logician Essenin-Volpin divided formal logical systems into two classes: democratic and totalitarian. In a democratic system, the rules tell you what is forbidden. By default, the rest is allowed. In a totalitarian system, the rules tell you what is allowed. By default, the rest is forbidden. (For those of you who understand only the language of categories, democratic and totalitarian systems are the final and initial objects of the appropriate category.) You wouldn't ask why Mr. A had not been allowed to go abroad. This would be a wrong question. You might ask why Mr. B had been allowed to go? That should have a good reason. For example, Mr. B might work for the secret police. Now you can see why the question "Why was formal logic forbidden?" is a wrong one. (Actually, they had a "reason" to forbid formal logic: the connection to philosophical positivism. Is positivism so exceptionally bad? Not necessarily. But it is certainly different from dialectical materialism, the only true philosophy.)

In the rest of this talk, logic means mathematical logic.

What is software technology?

This question is easy. We all know what hardware technology is. Software technology is the direct opposite of hardware, except it is a little harder.

Boom and gloom. Software technology is booming, but it goes through a severe crisis as well: reliability, compatibility, verifiability, etc. You name it. Some hackers do not realize that. They happily hack and change our world. They should be explained to that there is a severe crisis out there. The poor devils badly need guidance and organizing principles. This is where AMAST comes in.

What does "x AND y" mean?

The third question about AMAST was about the AND of AMAST. We stumbled also upon the meaning of "yes and no". Let us generalize and consider a more general question: what does "x and y" mean where x, y are arbitrary things (not statements)? Our discussions with learned colleagues turned up a couple of possible answers.

(1) The set {x,y}. This answer may be blatantly wrong. AM and ST = AMAST which isn't a set of two elements. The organizing committee, all by itself, has more than two elements.

(2) The fact that the intersection of x and y is nonempty. That sounds a little more convincing, but cannot be quite right because x and y are not necessarily sets.

Notice that in both answers, AND is commutative, which is not true in general. It is well known for example that the Communist founders are Karl Marx and Friedrich Engels, not Friedrich Engels and Karl Marx.[1]

(3) "x vs. y". This third answer is not necessarily true as well. For example, the relations between AM and ST are not adversarial; AM loves ST, and ST couldn't care less about AM.

One hazard of the trade: wrong abstraction level

It is clear by now that we overabstracted our third question. A wrong abstraction level is one of the greatest hazards of our trade.

If the abstraction level is too low, you have too many details. There are no theorems to prove or apply.

If you abstract too much, you may find yourself in a sterile atmosphere with no theorems (well, with only shallow theorems) to prove. Alternatively, this may be a delightful trap. You may find yourself in a very fertile atmosphere with numerous attractive theorems but this could be a problem too, because you may lose sight of the original question. For example, we find it very tempting to proceed with the investigation of the meaning of "x and y" in its full generality.

It may be in the eye of the beholder whether you abstract too much or not. For example, define programs equivalent if they compute the same thing, and find yourself in a delightful world of logic. Play with lambda calculus and types. (Didn't you really want to be a logician?) Ignore those silly programs that do not behave properly. The Unix kernel, for example. What does it compute? Nothing. It doesn't even converge. Modulo some side effects, it is equivalent to a trivial infinite loop.

What is the right level of abstraction? This is the art of our science. That is what AMAST is all about.

[1] At this point during the talk, Vaughan Pratt said, "See Paris and die".

So what does AND mean in the AMAST context?

Is it "motivated by"? There is a good precedent for this interpretation. That is what AND often means in the famous phrase "logic and computer science". We believe that "motivated by" isn't the main meaning in our case. As we mentioned above, there is an implication of [desired] applicability in the phrase "algebraic methodology".

Is it "applied to"? Hardly.

The most appropriate meaning seems to be: To be applied to [indirectly][eventually]. In other words, the meaning is "for".

Another delightful trap

You dive into mathematics and ... never come back. This trap is similar to but different from the one we discussed earlier.

For example, you write a book on Principles of Programming Languages. You have to give some formal semantics, of course. Denotational semantics seems fun. It requires domain theory though, and domain theory requires fixed-point theory. You explain all this carefully. The project goes along quite nicely. Suddenly, you panic! You have to say something about programming languages as well. A real language, like C, would be too much detail and trouble, this is obviously too low an abstraction level. You already gave the semantics of lambda calculus which is, all by itself, a programming language par excellence. How about the while language? Good. This should satisfy all those imperative freaks.

More on the AND of AMAST

There are of course other cases of "x and y" where AND means FOR. But there is something special about the AMAST use of AND. Consider, for example, the case when x = math and y = physics. Imagine you would like to apply some beautiful mathematics to some physics that does not quite fit your mathematics. What can you do? You can write science fiction but you cannot change the physical world. The situation is quite different if x = AM and y = ST. In principle, you can change ST. Why do they use those silly imperative languages that do not fit my mathematics? They would be much better off using functional languages or logic programming.

Future research

AMAST is in a business of changing the world of software technology. AMAST activities are hazardous, delightful and blessed with opportunities. They are approved and supported by the highest offices of the land like the Office of Naval Research.

Theorem 2 *AMAST is A MUST.*

Proof sketch At this moment, we can only give a very preliminary sketch of our proof. The next AMAST will be in Europe. In one of the dialects of Europese, AND is UND. This accounts for the crucial change of A to U. □

Acknowledgement I am greatly thankful to Neil Jones and Zoe Gurevich for patiently listening to successive versions of this talk, for useful comments and for encouragement.

Education Day

AMAST'93

Third International Conference
on
Algebraic Methodology and Software Technology

University of Twente
The Netherlands

Participants' Proceedings

Some tentative thoughts on teaching computer science

Hans-Jörg Kreowski
Universität Bremen
Fachbereich Mathematik/Informatik
Postfach 33 04 40
D-28334 Bremen
email: kreo@informatik.uni-bremen.de

1. Most of the students of today will be among the scientists, engineers, technologists, managers, teachers, politicians, etc. of the next 30 to 40 years. Hence teaching in universities in general and teaching computer science in particular are challenging tasks with high responsibility. What students learn, know and think and how they deal with it may form them to a good part and, in this way, may influence the future of science, technology, economy, politics, society, etc. I fear that not all university teachers are aware of this responsibility.

2. Teaching can be a hard job, in particular, if the teacher stands in front of an audience of 50, 100 or 500 students and has got only a vague idea of the levels of knowledge, motivation, interest and ability present. Frustration is not surprising under such circumstances, and enthusiasm seems to be wasted. Although the situation of teaching in universities needs a revisition (at least in Germany), there is still the chance of success from time to time because students acknowledge the effort of teachers as far as I can see. Teachers must try.

3. Clearly, teaching is much more than the repetition of knowledge found in books. Knowledge is only the basic material that needs proper combination, interpretation, cross references and, above all, the teacher's personal comments and views. The aim of teaching is not just to lecture on important matters to passive listeners, but to raise the students' interest, motivation and ability to play with, to work on, to think about and to understand the matter at hand actively and in their own fashion. University teachers must be good scientists and good animators.

4. Computer science is an engineering and scientific field in an embryonic state that is rooted in mathematics and electrical engineering. It is assumed to provide key technologies for the future development of economy and society (at least in the well-developed countries). The outcome of computer science is changing the work and life of many people. Hence teaching computer science must reflect the whole spectrum of relevant aspects from mathematics to social sciences. But how can this be achieved in an undeveloped field? A balance seems necessary between the well-understood basic matters of mathematics, engineering and social science useful in computer science and the urgent and actual questions that have got so weak and shallow answers up to now. But what is sufficient?

5. The trouble with teaching theoretical computer science is a bit different. There is the wealth of mathematics one can employ. There are already some fairly well-developed theories on basic objects of interest in computer science. But most of the students (at least those I know) do not enjoy mathematics, are not able to understand it properly or do not try hard enough. Hence motivation is mandatory. Unfortunately, a successful motivation is not very helpful if students understand the value of theoretical computer science, but are still not able to understand the matter itself.

Mathematics of Computation for (Software and Other) Engineers

David Lorge Parnas
Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

1 Preliminary Provocation

The title of this paper implies that Software Engineers are Engineers, i.e. that "software" plays the same role in their title that "Electrical", "Mechanical", or "Chemical" play in the titles of other engineering specialities. This, in itself, would seem to be a controversial statement, since it suggests that the model for software engineering education should be engineering education, not science education or mathematical education. That is my opinion, and one of the assumptions underlying this paper, but it is not the subject of this paper.

I like the term, "mathematical engineer", which I am told is used by some Dutch Technical Universities for software engineering. It seems to me that, just as certain areas of Electrical Physics comprise the basic knowledge of an Electrical Engineer, certain areas of mathematics, which includes (in my opinion) the most substantive areas of Computer Science, should be the basic knowledge that characterises software engineering. However, we should not forget that just as Chemical Engineers need to know much more than chemistry, Software Engineers will need to know more than Computer Science and Mathematics. Because we cannot teach them everything we think they should know, there might be some fundamental areas of Computer Science and Mathematics that we don't have time to teach them.

2 The role of mathematics in engineering

Those who do not have an engineering education themselves often fail to realise how much mathematics engineers learn. At my university, approximately 30% of an engineer's education is devoted to things that are explicitly titled mathematics. There is a great deal of mathematics taught in the specialised engineering courses as well. This is not atypical; it is often required by accreditation committees that control whether or not the graduates of a programme can easily be recognised as professional engineers.

Mathematics can be said tr be one of the things that differentiate professional engineers from technicians. A major emphasis in engineering education is the concept of professional responsibility. An Engineer is taught from her first day at University, that her products must be "fit for use". Engineering students learn that they cannot trust their intuition and "eyeballing" to be sure that a product is "fit for use". Their education is, in great part, devoted to learning how to do both mathematical analysis, and carefully planned testing, of their proposed designs. They also learn to accept, as completely normal, thr fact that their work will be subject to careful analysis and criticism, often based on mathematical analysis, by others. My own engineering education included approximately as much mathematics as would have been taken by a mathematics major and, at my *alma mater*, many of the courses were taken together with the mathematics majors. Regrettably, it is common to find special engineering mathematics courses, and to find that the mathematics professors who teach those courses assume that they are teaching people whose intellectual level is not as high as that of mathematicians. Having taught both, I do not see differences in ability, but I do see differences between the viewpoints of engineering students, and those of students majoring in mathematics or science.

Although engineers study a lot of mathematics, an engineer's view of mathematics is substantially different from that of mathematicians. Roughly put, engineers can take a lot for granted. Because their use of mathematics is always for the description and analysis of some physical product, they simply assume that functions have the properties that all functions describing physical products must have. They often do not bother to state those assumptions explicitly. This appears sloppy to many "formalists". In most cases, the mathematics is perfectly sound if one adds the assumptions explicitly in an environmental declaration. Because engineers are working in situations where it is clear which symbols in their equations are variables and what they represent, they do not see a need for explicit mathematical notations such as the lambda notation. Because they always know the range of values for their variables, and they know what they are trying to compute, they see little need for the quantifiers, type, and signature declarations that logicians demand of their colleagues. Whereas mathematicians are primarily interested in deep theorems and general properties of classes of expressions, engineers are often concerned with "junk" theorems and detailed anaiyses of special cases. In such situations the complex and careful habits of logicians seem quite unworkable and there is always a gap between a mathematician's treatment of a subject and that of an engineer who uses the same fundamental mathematics. What one chooses to record explicitly, the other tends to assume without much discussion. Those interested in exploring such issues further, should look at some of the writings of N.G de Bruijn and his students who had to pay a lot of attention to the "short-cuts" used by working mathematicians and engineers when they were developing their "Automath" system. [6]

It must also be recognised that the mathematics is often implicit, rather than explicit, in engineering notations. When an electrical engineer notes the inductance, resistance, and capacitance of a component, she knows that these are the parameters for a set of differential equations, but those equations are not always written down, just used when necessary. Again, we see that engineering notations take things for granted than a mathematician would want to see stated explicitly.

These remarks lead me to a pair of preliminary conclusions:

- Engineers, whether software or otherwise, can be expected to make extensive use of mathematics in the analysis of their products, including programs. Those who refuse to do so. are technicians, not engineers.
- When we develop mathematical methods for use by engineers, we need to respect the traditional differences between engineering mathematics and the type of mathematics promoted by "formalists" or logicians in the style of Hilbert. If we don't, we will be unnecessarily frustrated and quite ineffective.;

3 The role of programming in engineering

When I was an undergraduate, programming courses were optional. Moreover, no academic credit was given for them. The computer was considered to be a slightly enhanced version of the mechanical calculator. There was no more thought of including a computer course in the curriculum than we would think of including a course on the Marchand calculators that filled some laboratories, or a course on the slide-rules that many of us carried on our belts. It was expected that we would learn to use these "tools of our trade" on our own, or in non-credit courses. Programming was considered to be a simple mechanical task, "laying down instructions", akin to wiring up a circuit. Many engineers at that time had never taken a course in programming. When we began to offer the first credit course in programming at Carnegie Tech, there were many who feared that it would not have intellectual content analogous to a physics or calculus course. The Computer Science Department had to promise that they would not simply teach a programming language, but would teach something deeper and more lasting.

Today, things have changed - both for the better and for the worse. There is no longer any question about whether or not an engineer should have courses in programming. The computer, and software are now ubiquitous in engineering. Many engineering products include computers and software; many others are designed and analysed using computers. Hardly a week passes in which we do not hear some anecdote about the failure of an engineering product being caused either by the software contained in it or by an error in the software used to design it. Since people rarely talk loudly about their failures, we can assume that the anecdotes are just the "tip of the iceberg". Nobody questions the need for engineers to be good programmers and good at evaluating the software that they use.

However, there is something else that nobody questions any more: they do not question the intellectual content of many engineering courses in computing. Nobody asks whether the intellectual content of these courses is comparable to that of other math or science courses. Perhaps the question is not asked because the answer would be embarrassing. The typical course simply teaches a programming language, an artifact designed by one or more human beings. Most of the time is spent on things that are not mathematical truths, or even lasting truths; they are just design decisions by (often not very good) language designers. The courses are exactly equivalent to teaching about a particular calculator, including the location of its buttons, how to turn it on, how to change the display, etc. Many of these courses teach almost the same artifacts that were taught 30 years ago, but that is not the real problem. The real problem is that the <u>subject</u> of the course is the artifact. You can always tell that something is wrong when there is a big debate about which artifact to teach about. The situation is analogous to changing the lectures of a course on electrical circuit theory because we acquired new oscilloscopes. Another sure sign that something is wrong comes when someone defends a course by saying that they just introduced a <u>new</u> artifact.

We must also recognise another difference between engineering education and the education of scientists and mathematicians. In engineering schools there is a major emphasis on *design*. We are required by our accreditation committees to identify a large part of our curriculum as design. Design and analysis can be understood as complementary skills. Design is inherently creative and all that we can teach are heuristics, things that don't always work. Consequently, solid, disciplined analysis is necessary. The mathematics is taught as part of the analysis component of these courses, not the design. This is in sharp contrast to the attitudes taken by another famous Dutchman in our field. E. W. Dijkstra, and his followers, like to talk about mathematical derivations of programs from specifications. This is not the attitude taken in other areas of engineering. Design is recognised as a very creative task, in which mathematics and science provide essential inputs, but the primary role of the mathematics comes in the documentation and validation of the design. Program derivation from requirements appears analogous to deriving a bridge from a description of the river and the expected traffic. Refining a formal specification to a program would appear to be like refining a blueprint to a produce a bridge. Engineers always make a distinction between the product and the description of it. This seems to be lost in the computer science literature on programming and software engineering.

Those who chose engineering as a career path are often people with fairly a pragmatic view of life. They appreciate mathematics that is simple and elegant but they want frequent assurance that the mathematics is useful. It is important to show them how to use a mathematical concept, not simply to teach them the definitions and theorems. In engineering mathematics the emphasis has always been more on application of theorems than on proofs.

4 The mathematics needed for professional programming

I have recently taught a new course for first year engineers of all specialities. It replaced a course that could have been taught 30 years ago. I made two major changes:

- A large part of the course taught the basic mathematics behind programming with emphasis on the use of mathematics to describe what a program does, or must do, without giving an algorithm. All programming assignments were expressed as mathematical specifications.
- It was made very clear that the language was not the subject of the course. Students were given a choice of two programming languages that could be used in the laboratories. Two of the three lectures per week were taught in an algorithmic notation based on Dijkstra's guarded commands. The third, "laboratory", lecture taught a "real" language.

The course emphasised both the creative steps in programming and the analytical steps needed to confirm that one had not just created a monster.

The remainder of this section describes the mathematical contents of that course and how we used the mathematics to teach programming.

4.1 Finite State Machines

The first step in getting students to take a professional approach to programming is to get rid of the "giant brain" and "obedient servant" views of a computer. It is essential that students see computers as purely mechanical devices, capable of mathematical description, Students are taught that "remembering" or "storing" data is just a state-change, and taught to analyse simple finite state machines to "show" that they accomplish simple recognising tasks. The Moore-Mealy model is used.

4.2 Sets, functions, relations, composition

We present the basics of a naive set theory in which all sets consist of a finite number of elements from previously defined universes. We present the concept of relations (functions) as sets, and the operation of union, intersection, negation and functional (relational) composition. It is important to present the students with examples of the use of these concepts and exercises in the use. We want the students to know far more than the definitions and the algebraic laws; we ask them to apply the concepts to provide precise models of real-world situations. We show how the state machines that they learned about can be described by a pair of mathematical relations.

4.3 Mathematical Logic based on finite sets

In the first two sections, finite state machines, and sets have been kept not just finite, but small, so that they could all be described by enumeration. The next step is to point out that these are unrealistically small sets, that we cannot afford to describe most sets by enumeration, and that we must be able to make general statements about classes of states. We then introduce an interpretation of classical predicate logic in which all expression denotations are finite sets and we show them how to use predicate calculus to characterise sets, including functions and relations. The logic that we use allows partial functions (defining all primitive predicates on undefined values to be *false*). It is important to provide numerous examples in which the students use predicate logic to characterise the states of something real. Arrays (viewed as partial functions) provide a rich source of examples such as, "Write a predicate that is *true* if array A contains a palindrome of length 3." Again, it is important to show the use of the mathematics to say important things about programs, and to teach them to *use*, as contrasted to prove theorems about, logic. The interpretation of logic that we use is described in [1].

4.4 Programs as "initial states"

We provide a brief, and unconventional, view of programming as picking the initial state of a finite state machine. This is necessary when one wants to explain such concepts as table driven programs, interpreters, etc. At this point, I point out von Neumann's chief insight (in the area of computer design), the interchangeability of program and data.

4.5 Programs as descriptions of state-sequences

We then give a more conventional view of programs as descriptions of a sequence of state changes. Each program, given an initial state, describes one or more sequences of state changes. This concept is presented abstractly, we do not give any programming language notation for describing the sequences.

4.6 Programs as functions from starting-state to stopping-state

After pointing out that programs can be characterised as either *terminating* or *non-terminating* we indicate that this first course focuses on programs that are intended to terminate after computing some useful values. We then show that the most important characteristics of programs can be described by a mathematical relation between the starting-states and the stopping states. The exact model used, LD-relations, is described in [2] or [3]. Here too, it is essential to provide examples in which the students use relations to describe distinct sets of sequences that are equivalent in the sense of having the same set of (start-state, final-state) pairs.

4.7 Tabular descriptions of functions and relations.

We extend the notation of predicate calculus by introducing 2-dimensional tableaux, which we call simply tables, whose entries are predicate expressions or terms. We show that these are equivalent to more conventional notation, but easier to read. Students are given many examples in which we describe mathematical functions using these tables [7]

4.8 Teaching programming with this mathematical background.

The remainder of the course is devoted to teaching students to program. All programs are introduced, not with a natural language description, but with a mathematical description of the required behaviour. The simple programming notation that is used (essentially that in [3]) is defined using the mathematical concepts above. We begin with very simple programs and continue, always using the same discipline to cover more complex engineering problems. Homework assignments are given using the tabular notation. Students are shown how to systematically determine if a program in this notation covers all cases and does the right thing in each case. Although, we never talk of "correctness proofs" we do use correctness concepts to explain a program. For example, we usually identify an "invariant" when explaining a loop, and use a monotonically decreasing quantity to convince students that a program will terminate.

5 The mathematics needed for software engineering

For many years I have taught courses entitled "Software Engineering" usually to students in the third or fourth year of university. Although the course has a significant design and pragmatic content, it has also been necessary to teach some mathematical concepts. Generally, Computer Science students have inadequate mathematical preparation for the course; they have learned too much theoretical computer science and too little about fundamental mathematics. However, the preparation of my Computer Engineering students seems even worse. They have had lots of mathematics, but the wrong mathematics. In this section, I will describe the mathematical basis of my software engineering class. The class covers the "standard" software engineering topics and students are asked to do practical exercises, but the basic message is that they must produce a sequence of documents whose contents must be representations of key mathematical functions. The approach is basically that in [4]. To get maximum benefit from the course, students should already be familiar with the concepts described in the previous section. Usually, they have not had the necessary exposure, and much of the course must be devoted to mathematics.

5.1 How can we document system requirements?

A critical step in documenting the requirements of a computer system is the identification of the environmental quantities to be measured or controlled and the representation of those quantities by mathematical variables. The environmental quantities include: physical properties (such as temperatures and pressures), the readings on user-visible displays, administrative information, (such as the number of people assigned to a given task), and even the wishes of a human user. These must be denoted by mathematical variables in the way that is usual in engineering. That association must be carefully defined, coordinate systems, signs etc. must be unambiguously stated.

It is useful to characterise each environmental quantity as either monitored, controlled, or both. *Monitored quantities are those that the user wants the system to measure. Controlled* quantities are those whose values the system is intended to control. If needed, time can be treated as a monitored quantity. In the sequel, we will use "m_1", "m_2", ..., "m_p" to denote the monitored quantities, and "c_1", "c_2", ..., "c_q" to denote the controlled ones. Because it is often the case that a system is intended to both monitor and control certain quantities, these lists might have variables in common.

Each of these environmental quantities has a value that can be recorded as a function of time. When we denote a given environmental quantity by "v", we will denote the time-function describing its value by "v'". Note that v' is a mathematical function whose domain consists of real numbers; its value at time t is denoted by "$v'(t)$".

The vector of time-function $(m'_1, m'_2, \dots, m'_p)$ containing one element for each of the monitored quantities, will be denoted by "$\underset{\sim}{m}'$"; similarly $(c'_1, c'_2, \dots, c'_q)$ will be denoted by "$\underset{\sim}{c}'$".

5.1.1 Relation NAT

The environment, i.e. nature and previously installed systems, place constraints on the values of environmental quantities. These restrictions may be documented by means of a relation, which we call NAT. It is defined as follows:

- domain(NAT) is a set of vectors of time-functions containing only the instances of $\underset{\sim}{m}'$ allowed by the environmental constraints,

- range(NAT) is a set of vectors of time-functions containing only the instances of $\underline{c}'$ allowed by the environmental constraints,
- $(\underline{m}', \underline{c}') \in$ NAT if and only if the environmental constraints allow the controlled quantities to take on the values described by $\underline{c}'$, when the values of the monitored quantities are described by $\underline{m}'$.

NAT is not always a function; if NAT is a function the computer system will not be able to vary the values of the controlled quantities without effecting changes in the monitored quantities.

5.1.2 Relation REQ

The computer system is intended to impose further constraints on the environmental quantities. The permitted values may be documented by means of a relation, which we call REQ. It is defined as follows:

- domain(REQ) is a set of vectors of time-functions containing the instances of $\underline{m}'$ allowed by environmental constraints,
- range(REQ) is a set of vectors of time-functions containing only those instances of $\underline{c}'$ considered permissible,
- $(\underline{m}', \underline{c}') \in$ REQ if and only if the computer system may permit the controlled quantities to take on the values described by $\underline{c}'$, when the values of the monitored quantities are described by $\underline{m}'$.

REQ is usually not a function because the application can tolerate "small" errors in the values of controlled quantities.

5.1.3 Requirements feasibility

Because the requirements should specify behaviour for all cases that can arise, it should be true that,

(1) $\text{domain(NAT)} \subseteq \text{domain(REQ)}$.

The relation REQ can be considered *feasible with respect to NAT* if (1) holds and

(2) $\text{domain(REQ} \cap \text{NAT)} = (\text{domain(REQ)} \cap \text{domain(NAT)})$.

Feasibility, in the above sense, means that nature (as described by NAT) allows the required behaviour (as described by REQ); it does not mean that the functions involved are computable or that an implementation is practical.

Note that (1) and (2) can be reduced to:

(3) $\text{domain(REQ} \cap \text{NAT)} = \text{domain(NAT)}$.

5.2 How can we document system design?

During the system design two additional sets of variables are introduced: one represents the *inputs*, quantities that can be read by the computers in the system; the other represents the *outputs*, quantities whose values are set by the computers in question. These variables are associated with input and output registers on the computers in the system; their values will also be described by time-functions.

In the sequel we assume that $\underline{m}'$ and $\underline{c}'$ are defined as in Section 4.2.

5.2.1 Relation IN

Let "$\underline{i}^t$" denote the vector $(i_1^t, i_2^t, \ldots, i_r^t)$ containing one element for each of the input registers. The physical interpretation of the inputs can be specified by a relation IN, defined as follows:

- domain(IN) is a set of vectors of time-functions containing the possible instances of $\underline{m}^t$,
- range(IN) is a set of vectors of time-functions containing the possible instances of $\underline{i}^t$,
- $(\underline{m}^t, \underline{i}^t) \in$ IN if and only if $\underline{i}^t$ describes possible values of the inputs when $\underline{m}^t$ describes the values of the monitored quantities.

IN describes the behaviour of the input devices. It is a relation rather than a function because of imprecision in the measurements. It must be the case that,

$$\text{domain(NAT)} \subseteq \text{domain(IN)},$$

because the input device must transmit some value for every condition that can occur in nature.

5.2.2 Relation OUT

Let "$\underline{o}^t$" denote the vector $(o_1^t, o_2^t, \ldots, o_s^t)$ containing one element for each of the output registers. The effects of the outputs can be specified by a relation OUT, defined as follows:

- domain(OUT) is a set of vectors of time-functions containing the possible instances of $\underline{o}^t$,
- range(OUT) is a set of vectors of time-functions containing the possible instances of $\underline{c}^t$,
- $(\underline{o}^t, \underline{c}^t) \in$ OUT if and only if $\underline{c}^t$ describes possible values of the controlled quantities when $\underline{o}^t$ describes the values of the output quantities.

OUT describes the behaviour of the output devices. It is a relation rather than a function because of device imperfections.

5.3 How can we document software requirements?

The software requirements are determined by the system design document and the system requirements document. As mentioned earlier, the *software requirements document* can be seen as a combination of those two documents. It would contain the relations NAT, REQ, IN, and OUT.

In the sequel we assume that REQ is feasible with respect to NAT, and that $\underline{m}^t$, $\underline{c}^t$, $\underline{i}^t$ and $\underline{o}^t$ are defined as in previous sections.

5.3.1 Relation SOF

The software will provide a system with input-output behaviour that can be described by a relation, which we call SOF. It is defined as follows:

- domain(SOF) is a set of vectors of time-functions containing the possible instances of $\underline{i}^t$,
- range(SOF) is a set of vectors of time-functions containing the possible instances of $\underline{o}^t$,
- $(\underline{i}^t, \underline{o}^t) \in$ SOF if and only if the software could produce values described by $\underline{o}^t$ when the inputs are described by $\underline{i}^t$.

SOF will be a function if the software is deterministic.

5.3.2 Software acceptability

For the software to be acceptable, SOF must satisfy[1]:

(1) $$\forall \underline{m}^t\ \forall \underline{i}^t\ \forall \underline{o}^t\ \forall \underline{c}^t [\mathrm{IN}(\underline{m}^t, \underline{i}^t) \wedge \mathrm{SOF}(\underline{i}^t, \underline{o}^t) \wedge \mathrm{OUT}(\underline{o}^t, \underline{c}^t) \wedge \mathrm{NAT}(\underline{m}^t, \underline{c}^t) \Rightarrow \mathrm{REQ}(\underline{m}^t, \underline{c}^t)] \quad .$$

Note, that if one or more of the predicates IN($\underline{m}^t$,$\underline{i}^t$), OUT($\underline{o}^t$,$\underline{c}^t$), or NAT($\underline{m}^t$,$\underline{c}^t$) are false, then any software behaviour will be considered acceptable. For example, *if a given value of* $\underline{m}^t$ *is not in* the domain of IN, the behaviour of acceptable software in that case is not constrained by (1).

If we assume that relations REQ, IN, OUT, and SOF are functions, we can use functional notation to rewrite (2) as follows:

(1a) $$\forall \underline{m}^t [\underline{m}^t \in \mathrm{domain}(\mathrm{NAT}) \Rightarrow (\mathrm{REQ}(\underline{m}^t) = \mathrm{OUT}(\mathrm{SOF}(\mathrm{IN}(\underline{m}^t))))]$$

The writers of the requirements document must describe the relations NAT, REQ, IN, OUT. The implementors determine SOF and verify (1) or (1a). A document of this type may require natural language in the description of the environmental quantities, but can otherwise be precise and mathematical. The use of natural language in the definition of the physical interpretation of mathematical variables is unavoidable and quite usual in engineering.

5.4 How can we document software behaviour?

Although the software requirements document fully represents the requirements that the software must meet, it may allow observable differences in behaviour. It will often be desirable to specify a subset of the behaviours allowed by the requirements document for actual implementation. In this way designers will make certain decisions that might otherwise have been left for the programmers. The relation SOF can be described in a separate document known as the *software behaviour specification*. This document is especially important for multiple-computer systems because it will define the allocation of tasks to the individual computers in the system. For computer networks, or multi-processor architectures one may see a hierarchy of software behaviour specifications with an upper level document assigning duties to a group of computers, and the lower level documents detailing the responsibilities of smaller groups of computers. The lowest level documents would describe the behaviour of software for individual computers.

5.5 How can we document black-box module interfaces?

Most modern computer systems require software of such size and complexity that it cannot be completed by a single person in a few weeks. For many reasons it is desirable to decompose the software construction task into a set of smaller programming assignments. Each assignment is to produce a group of programs (cf. Section 4.8) which we call a *module*. We view each module as implementing one or more finite state machines, frequently called *objects* or *variables*. A description of the module interface is a black-box description of these objects.

Writing software module interface specifications is similar to documenting software requirements but some simplifications are possible. Many software modules are entirely internal; there are no environmental quantities to monitor or control and all communication can be by means of external invocation of the module's programs. Moreover, the state set of a software module is finite, and state transitions can be treated as discrete events. For most such modules, real-time can be neglected because only the sequence of events matters. This allows us to replace the general concept of time-function by a sequence describing the history in terms of discrete events; we call these sequences *traces*.

[1] In the following the universes from which $\underline{m}^t$, $\underline{c}^t$, $\underline{i}^t$ and $\underline{o}^t$ are drawn are assumed to include all vectors of time-functions.

We identify a finite subset of the set of possible traces, which we call *canonical traces*. Every trace is equivalent[2] to a single canonical trace. *Trace assertion specifications* comprise three groups of relations:

(1) Functions whose domain is a set of pairs (canonical trace, event) and whose range is a set of canonical traces. The pair $((T_1,e), T_2)$ is in the function if and only if the canonical trace T_2 is equivalent to the canonical trace T_1 extended with "*e*". These functions are known as *trace extension functions*[3].

(2) Relations whose domain contains all the canonical traces and associate each canonical trace with a set of values of output variables.

(3) Functions whose domain is the set of values of the output variables and whose values define the information returned by the module to the user of the module.

5.6 How can we document internal module design?

Each module has a private data structure and one or more programs. We propose to document the design sufficiently precisely that its correctness can be verified. The internal documentation of a module contains three types of information:

(1) A complete description of the data structure, which may include objects implemented by other modules.

(2) A function, known as the *abstraction function*, whose domain is a set of pairs (object name, data state), and whose range is a set of canonical traces for objects created by the module. The pair ((*on*, *ds*), *T*) is in this function if and only if a trace equivalent to *T* describes a sequence of events affecting the object named *on* that could have resulted in the data state *ds*.

(3) An LD-relation [2,3], often referred to as the *program function*, specifying the behaviour of each of the module's programs in terms of mappings from data states before the program execution to data states after the execution

6 Do we need new mathematics or merely new representations?

There is something in the above that will be disturbing, perhaps even annoying, to many people. We have managed to make precise mathematical statements about software engineering using classical mathematical concepts. We have not used any of the relatively new "specification languages", which have been developed especially for software engineering applications. We have even been able to talk about the real-time characteristics of systems without introducing any changes in our logic for that purpose; we have dealt with real-time using the traditional engineering approach, the use of functions whose range and domain are taken from the set of time-functions. I have studied the new "formal methods" and simply do not see how they add value. It seems to me that the mathematics needed by engineers to understand software is very close to the classical mathematics that was developed before Computer Science became an identified "discipline". In [5] I presented some serious doubts about the direction taken by Computer Science; this paper presents further grounds for those doubts.

On the other hand, when we tackle real software engineering problems, such as the A-7 Onboard Flight Program [8], or the Darlington Nuclear Plant [9], we find a need, not for new basic

[2] Two traces are *equivalent* if they have the same effect on future behaviour of the object.

[3] A trace extension function is sometimes called a *reduction* function.

concepts but for new notations. The use of conventional, one-dimensional, notation to describe functions and relations resulted in pages of repetitive formulae that were hard to parse. It is for this reason that we have introduced the multidimensional notations, first used in [8] and described in [7]. If the new specification languages are new, it is only in their semantics, they have deviated in no significant way from the one-dimensional notation that is traditional in mathematics. Our experience suggests that the semantic issues are not the serious ones. New notation, with classical semantics, has proven very practical.

7 Acknowledgements

These thoughts have been strongly influenced by H. D. Mills and N.G. de Bruijn. Some of the text was taken from a paper written jointly with Prof. Jan Madey of Warsaw University ([4]).

8 References

[1] Parnas, D.L., "Predicate Logic for Software Engineering", *CRL Report 241,* McMaster University, TRIO (Telecommunications Research Institute of Ontario), February 1992,. Accepted by IEEE Transactions on Software Engineering.

[2] Parnas, D.L., "A Generalized Control Structure and Its Formal Definition", *Communications of the ACM,* Vol. 26, No. 8, August 1983, pp. 572-581.

[3] Parnas, D.L., Wadge, W.W., "Less Restrictive Constructs for Structured Programs", *Technical Report 86-186,* Queen's, C&IS, Kingston, Ontario, Canada, October 1986,

[4] Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering (Version 2)", *CRL Report 237,* McMaster University, TRIO (Telecommunications Research Institute of Ontario), September 1991, 14 pgs

[5] Parnas, D.L., "Education for Computing Professionals", *Proceedings of International Conference on Computing and Information,* ICCI'90, Niagara Falls, Ontario, May 23-26, 1990. Published in *Advances in Computing and Information,* S.G. Akl, F. Fiala, W. Koczkodaj (editors), Canadian Scholars' Press Inc., 1990, pp. xi (ISBN 0-921627-70-X).

[6] Nederpelt, R.P., "De taal van de wiskunde", 1987, Versluys Uitgeverij bv -
Almere, The Netherlands.

[7] Parnas, D.L., "Tabular Representation of Relations", *CRL Report 260,* McMaster University, TRIO (Telecommunications Research Institute of Ontario), October 1992

[8] Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., "Software Requirements for the A-7E Aircraft", *NRL Memorandum Report 3876,* United States Naval Research Laboratory, Washington D.C., November 1978, 523 pp.

[9] Parnas, D.L., Asmis, G.J.K., Madey J., "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety,* Vol. 32, No. 2, 1991, pp. 189-198.

Mathematical training for the software developpers:

A practical experience

Jacques PRINTZ
SYSTAR
171 Bureaux de la Colline
92213 Saint-Cloud Cedex
France

1. Introduction

What kinds of mathematics are usefull for the software developpers raise the general question of what is software in a quite similar way that if we ask what kinds of mathematics are usefull for the chemists or the biologists.

Arguing on the very nature of software might rapidely become a rather academical and artificial question without any consistent answer. Following the advice of Wittgenstein "Don't ask for a meaning, ask for a use!" I will prefer starting from the use.

In the case of software engineeering, the basic question is: What kind of information systems are we trying to build today, and, inside these systems, what is the role devoted to the software?

At the early time of J.Von Neumann and until the mid 70th's software was mainly a problem (sometimes very difficult) of creating algorithms. Consequently the way to express algorithms, that is to say the issue of having "good" programming languages, was ones of the dominant questions.

The amount of software development of that time was the production of small sized staff, often reduced to a single designer and some programmers, but with the interesting caracteristic of having being well trained in mathematics,

- either thru numerical analysis for practical computation problems as monte carlo method or operations research or statistics,...
- or, more rarely, thru mathematical logic for creating sound system architecture, computation model or system model description including linguistic aspects,...

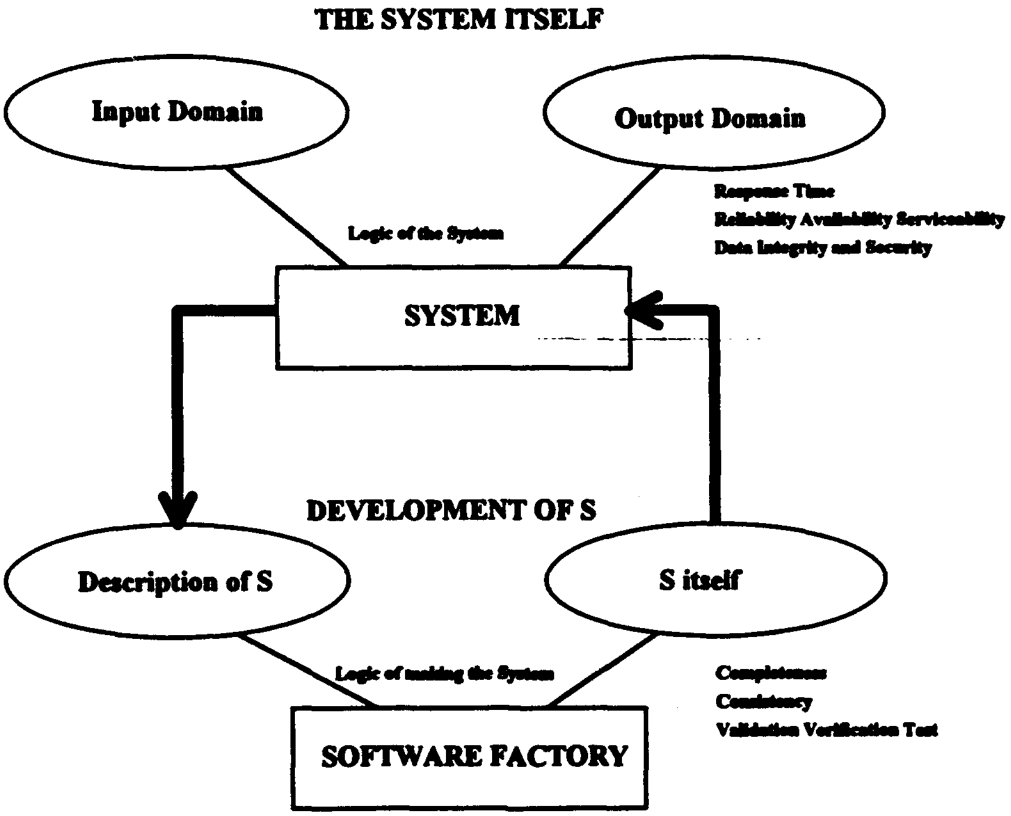
A good deal of mathematics has been elaborated and adapted at that time, mainly based on the outcomes of reseach in the field of mathematical foundation accumulated in the first half of the century. The strong connection between the early computer science and mathematical logic has given us the foundation of theoritical computer science and the mathematics associated with it: automata theory, formal languages, computability,... etc.

That theoritical area of knowledge has had a first direct practical application in the domain of programming languages and their associated translators. Reliable compiler construction is probably the most well known success and I can witness of it.

Starting from the early 80th's with the PCs revolution, the nature of software has been progressively and completely modifyed. It is becoming massiv and is better caracterized by a more or less depth entanglement in large or very large systems, in the sense of General System Theory, where some parts of the system are software and some others are devices of any types which may include human beings to perform computation still beyond the capacity of machines (as for example complex pattern recognition) or to take the appropriate decisions and control the system. In such systems each part influences the others, creating the so called "strange loops" whose side effects are to exponentiate complexity. Progressively, software is becoming "reactive" or embedded! The traditionnal opposition between scientific or real time software, and business software, i.e. Cobol software, is becoming meaningless with the rise of networks and graphical user interfaces.

Software is no more a solitary production and requires now large staff, sometimes several hundredth of developpers and years of development, whose global behaviour may be far of the elementary behaviour of its individual members in such a way that team organisation is becoming a major issue.

Professor Lehman, in his book "System evolution: the process of software change", has emphazised the strange relationship and duality which exist between the system to built and what he calls the meta-system, that is to say the software production system itself; but he hasn't provided any real explanation of what he has observerved so that what he has called " Law of software engineering" may only be considered as an experimental evidence. The situation may be depicted as follow:



It is now clear that understanding software implies necessarily to understand the global system context, not only the architectural aspect of it but also the process to built it: a kind of software embriology. Human aspect of software engineering is a major issue facing the certitude that it will be much more difficult to automate than any other engineering field such as hardware.
Again, a good deal of mathematics exists to describe the way systems behave: graph theory, operations research, theory of games, coding and information theory, modelization, etc...

Coming back to Von Neumann, I mentionned above, it is quite remarkable that in the latest part of his life when he literaly founded the theory of automata, he was especially interested to find practical solutions to what he considered as the three fundamental factors limiting the engineer's ability to build powerfull computer:

- the size of elementary hardware components,
- the reliability of the elementary components,
- the lack of a theory of logical organization of complicated system of computing elements.

Transposed in modern software terminology, we have the three basic issues:

- the size of elementary automata (as a mathematical model of programs),
- the reliability of automata (as an elementary proof of syntaxic and semantic correctness of programs),
- the way to group automata to form very large sets of cooperative automata, or, in other words, the way to organized them in order to be able to predict in a deterministic way, their global behaviour and their expected global reliability.

These three issues are the hard core of main interrogations for the professionnal, or at least mine! in order to offer a minimal warranty of the effectiveness of software engineering.

It is clear for me, and I hope for all of us, that, as in all the other engineering fields, mathematics will play a prominent role in future software engineering. Not only "pure" mathematical logic, but also all the mathematics mathematicians as Von Neumann, Turing, Ulam,... considered usefull and which are even more relevant whith our day to day problems.
It is important to note immediatly that some of the observed phenomena will deal with rigid, all-or-none concepts, which is the caracteristic of logic and that some others are better approximated with continuous concepts as for example reliability, serviceability and adaptability of very large systems.

From a pedagogical point of view, an this is a fundamental issue for software developper mathematical training, formal logic is one of the most refractory and abstract part of mathematics as well as a very recent one; so we have to consider the role of continuous model as an approximation of dicrete one's because continuous mathematics is the best cultivated portion of mathematics with the most historical background which provide us with large fields of interpretations and reformulations of classical questions. This methodological advice of JvN formulated years ago is still applicable.

However, it is not my intention to play the historian and present the prominent role of JvN in computer science, there is some good books on that subject. Reading master's work is still exciting and rarely a waste of time. So, returning to our subject of mathematical training, I will present briefly three problems I have been confront permanently during my professionnal life and the kind of mathematics one can guess behind them.

These three problems may be summarized as follow.

Problem # 1: We observe a great variability of the amout of programming required for systems intuitively perceived as very similar to an other one (a range of 1 to 10 may be easily observed). Is there anything similar in the field of mathematics and if yes, what is the explanation? How can we explain that slight variations in the specification of a system may create a totaly non-linear one at the programming (or automata) level in both direction, positive or negative? Do we have models for that or is there only chaos?

Problem # 2: Very large software systems (up to several million lines of code cannot be build from scratch. They require numerous intermediary steps before to be completed and fully operational. The question is: what is the dynamic of growth of such software systems? Wat is the complexity level one can manage step by step in order to avoid system construction divergence or oscillations. What is the amount of ancillary work to provide in order to bring the system in existence: a kind of thermodynamics second principle applied to software engineering!

Problem # 3: Very large systems with billions of states are far beyond our ability to provide formal proof of syntax and/or semantic correctness. What do we call *the proof* of such a system, who give us warranty that the proof is correct and much shorter than the sytem to prove? In other word, to speak as JvN, who custodes the custodies! Thus, system reliability becomes a matter of probability and statistics. The question is: what are the necessary conditions to ensure that the minimal has been done and what are, if any, the mathematical models to ensure that error effects will be kept under a minimal threshold given in advance as for example: the system may fail, but it must restart in a correct state in less than *x* second? This time this is an equivalent Shannon's second theorem which must be set up!

2. P#1 - The variability of software system size

It is extremely difficult, even impossible, to have a practical experience of software size variability in a classical software developper curriculum. To observ interesting phenomena, large amount of development is required, generaly incompatible which the kind of work which is asked to a student.

By chance, there is an interesting analogy between software development and mathematical development so that we can use mathematical development as a substitute to program development. The length of the proof of some theorems may vary in a wide range according to the way the theory has been settled: kind of objects, representation of the objects, choice of the axioms, and so on ...

A remarkable fact is that software size is weakly dependent of the programming language but highly dependent of the architecture and organisation of the whole software system. Similar situations exist in what mathematicians call local and global considerations in mathematical development [see A.Lautman: Essai sur l'unité des mathématiques].

Thus, the central thesis for P#1 is that there exist a pertinent analogy between program development and mathematical theory development so that studying the latest will provide us with insight for a better understanding of the former.

To go deeper, a brief recall of formal system theory will be needed. Very good books exist on formal systems and everything can be easily find.
At a first and rather intuitive level, we will consider a formal system as having 3 basic components, as follow:

Syntax of the formal system.

Syntax of a formal system, later on abbreviated FS, is well illustrated by what is called concrete syntax in programming language theory. This is a set of rules which explains how basic objects and elements of the system are settled; how more complex expressions may be built starting from the basic ones. Rules of naming - proper names and class/generic names - (of exceptional importance in any complex system) belong to the syntax of the FS.

Semantic of the formal system

As opposed to syntax which is purely abstract, semantic deals with *meaning*, that is to say how expressions may preserve properties as for example those of being *true* or *false* with regard a given domain into which they can be interpreted or translated. Again, programming languages allow us to illustrate simply what semantic is (but FS semantics is far beyond programming language semantics, so beware of a limited understanding!).
Strongly related to syntax and semantic is the distinction made by the logician between *intension* and *extension*. Intension deals with the form or syntax of an expression or a function [see G.Frege - Begriffschrift - for a detail and precise analysis]. Extension deal with the domains associated with the function, i.e input domains, output domains, state domains, error domains, etc...Intension and extension are in a dual relationship and are two ways of speaking of the same thing; intension and extension consideration are of exceptional importance in distributed systems where data and algorithms may be freely exchanged.

Pragmatic of the formal system

Pragmatic refers to the way the FS is used by the observer. In a logical perspective it deals with different interpretations which can be associated with the FS and how facts of the real world may be precisely associated with abstract domains defined in the FS. Obviously only a small subset of the facts perceived in the real world may be abstracted and associated with classes of the FS; such facts will received proper names to be identified unambiguously. Sciences as physic witness of the difficulty to assign meaning to abstract entities and to identify interesting abstract entities. In the programming world, pragmatic is of utmost importance (much more important than in mathematics) because we are interested to know how programs or systems behave and how they can be executed on real machines because we have to interact more and more with them. Ada language introduce a notion of *pragma* which is effectively relevant to pragmatic but which is far to cover all the pragmatic aspects associated with the program text. Difficulties of using Ada in hard real time systems with exact time constraints is a matter of language pragmatic; deterministic or non deterministic run time environment is an other one which is an implementation choice.

[...]

3. P#2 - Pattern of growth of software systems

Understanding the way and the conditions [sufficient and/or necessary] under which software systems can growth is a major issue for the information technology industry. It raises immediately two fundamental questions:

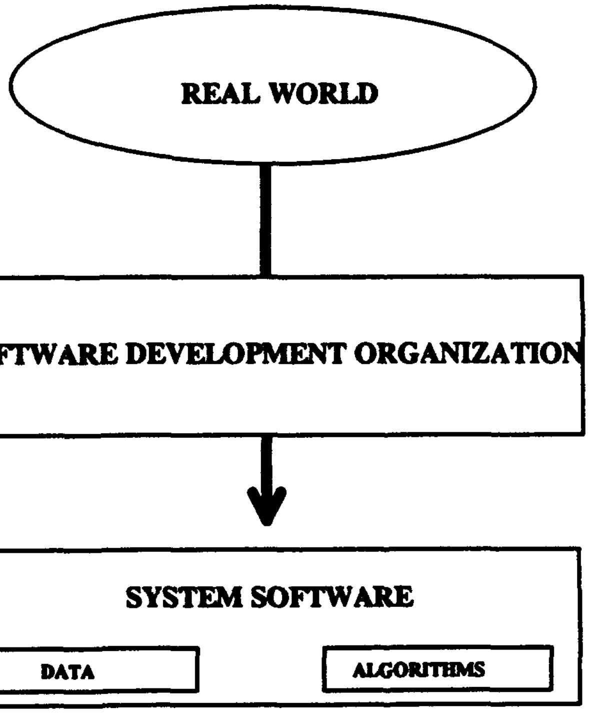
1. How can we measure or estimate software size, what is the unit of size?
2. Is there any limit to the size of a software system? Are there any limitation factors? If such factors exist, are they absolute or relative to a given maturity level of the software industry?

In the absence of well defined unit of size (this is the case, up to now!) models which can be built will have a strong qualitative taste and this is already an important limitation if we compare this situation with the state of the art of other scientific engineering fields.

Once a model is defined, another important topic is the dynamic of the model. Every one which has had the chance to work with large staff has been confront with some very strange phenomena as oscillations or instability which may cause dramatic system regression. The question is: what is behind? Is there anything in the software process which looks like dynamic instability similar to what we find in chaos? What is the effect of the arrow of time?

Thus, the central thesis for P#2 is that system dynamic is a fundamental topic of software engineering which need to be investigated in detail - that is to say with the help of models, even if they are qualitative - if we want to get a chance to understand factors which limit software productivity.

To sustain the thesis, I will present 3 elementary models and will give some explanations on how they relate to the real software world. Real word software development may de depicted as follow:



Parts of the real world are progressively translated in an executable software system. We are interested to know the efficiency of the transformation, in particular how the productivity ratio evolve according to the system structure and to the organization of the development.

Model 1 - M1 - is simply the exponential growth model when there is no limitation to the growth. The characteristic equation of the model is the classical one:

$$\Delta S = \varepsilon\, S\, \Delta T$$

where S is the actual size, ΔS the increase of the size, ΔT the increase of an abstract time (approximatively the amount of effort) and ε the rate of increase of S per unit of abstract time T.

Model 2 - M2 - is the *S* curve (also called the logistical curve) model of paramount importance in software engineering as in other engineering fields as chemical engineering or population dynamic. The well known equation of the model is

$$\Delta S = (\varepsilon - \lambda S)\, S\, \Delta T$$

where λ is a limiting factor which depend on the system structure whose effect is to diminish the rate of increase which is no more constant. This well known model is typical of growth in the context of limited resources.

Model 3 - M3 - is a little bit more sophisticated and take in account the organizational environment which may also induce additional limitation. As everybody knows, organization may become less efficient (and sometimes very badly) when they grow old. The corresponding term, called self-infection or self-destruction term, will have the form

$$\int_0^T S(u)F(u)du$$

into which F is a function which relates to the organization and its ability to generate noise which will reduce its efficiency. I will describe that function by giving an intuitive description of it with the help of a game theory model known as the prisonner dilemma.
The general equation of the model has now the following form

$$\Delta S = \left(\varepsilon - \lambda S - \int_0^T S(u)F(u)du\right) S\, \Delta T$$

which has been studied and integrated by Volterra [see the Volterra's classic: Théorie Mathématique de la Lutte Pour La Vie].

[...]

4. P#3 - Reliability of large software systems

It is a law of nature that reliability in a very broad sense deals with redundancy. Human language is highly redundant and so are brain organization or mamals DNA molecules. Hardware reliability, much more closer to us, is a supporting evidence that desesperate situation (remember what hardware technology was at the time of JvN) can be dominated and mastered. A prerequisite is that error phenomenon be recognized as a central question in software development, if not THE unique one, as it has been in other field as data transmission. To quote R.Hamming in his classic Coding and Information Theory "Most bodies of knowledge give errors a secondary role, and recognize their existence only in the later stages of the design. Both coding and information theory, however, give a central role to errors (noise) and are therefore of special interest, since in real life noise is everywhere".

Returning to our logical model roughly describes in P#1, errors may be present in the

- syntactic level,
- semantic level,
- pragmatic level.

Mathematics can give us significant help for the syntactical level by providing for that level well founded abstract object as automaton or elementary date structures which play for information representation the same role as real numbers and functions in other engineering fields.

Mathematics can give us some help for the semantical level by providing well founded domain definitions to which abstract objects belong and well founded transformation rules from one domain to an other one. Abstract monitor models as CSP or concurrency models or programming models or normal forms in data modelling belong to that level.

Mathematics is of limited help, for not to said of no help, for the pragmatical level for which there is no alternative to verification and validation technics. Again this situation is rather common in all the engineering fields: there is no demonstration that the space shuttle is bug free or that a bridge will not break. Trials must be done. Worst situations occur when trials can't be done as for the Star War software system.

Thus, the fundamental issue is twofold:

First, the type of mathematical proofs which can be reached and in particular the complexity of the proof itself. Proofs must be constructive; if they are not much less shorter than the programs to prove, they are useless in the real engineering world. I consider as very promising the kind of correctness proofs done for VLSI whose main result is to dramatically reduce the simulation time to verify and validate the circuit.

Second, the management of redundancy to be added in the programs with a double question: how much redundancy? and where to insert it in the programs? The problem is how to reduce the time between the fault occurence and the fault detection by an observer and how to control the program overhead in such a way that the real time behaviour be kept under a threshold given in advance. Observer - a kind of Maxwell's deamon - gives us information of the state of the system but also modifies and adds uncertainties to the system behaviour as in Quantum Mechanics.

Then, the central thesis for P#3 is that redundancy management be recognized as the most important topic of software reliability and that information theory provide a conceptual framework to formulate clearly the very nature of information, redundancy and organization as well as the role of the human or artificial observer.

In that paper I will only sketch a research direction, focused on elementary behaviour of program flow (with the help of regular expressions) and on dynamic structure of the data references associated with the program (topological relations between both). By the way, we will see how test strategies may be make more effective.

[...]

5. Conclusion

Among the 3 problems presented in this paper, the latest is probably the most challenging for the future of information technology. With regard the age of software technology we are still in the period where we can simply and modestly observ the phenomenon. We certainly need to measure much more quantitatively what we observ and this is a preliminary condition to any progress. Software reliability ultimate aims will probably take much more time to be achieved than expected (as compared whith hardware engineering for which it has taken almost 50 years). To quote JvN " The great progress in every science came when, in the study of problems which were modest as compare with ultimate aims, methods where developed which could be extended further and further. ... . The sound procedure is to obtain first utmost precision and mastery in a limited field, and then to proceed to another, sometime wider one, and so on. ... . The experience of more advanced sciences, for example physics, indicates that impatience merely delays progress, including that of treatment of the *burning* questions. There is no reason to assume the existence of shortcuts".

Returning to the initial interrogations of usefulness of the mathematics for the software developpers, I will insist a last time on two aspects which seems to me of equal importance:

- First aspect is that there will be no future for software engineering without the help of mathematical methods and especially those of the discrete mathematics.
- Second aspect is that the way mathematical development is achieved thru the history, is of exceptional pedagogical importance.

Mathematical development has evolved thru the ages, new concepts have been added, formulations of classical problems have been entirely reformulated in a much more natural and elegant way, and so on. Logic is a good illustration of both aspects, although geometry and algebra offer probably far-reaching examples but may require more mathematical skill.
These rather aesthetical considerations seem to me of considerable importance for the software developpers, whose programs are (or should be) populated with abstract entities sometimes far beyond of intuitive evidence, by providing them with logical forms and reasonning schemes which are the foundation of rational, unambiguous and explicit thinking as well as reliable human communication.

References

[...]

Invited Papers

AMAST'93

Third International Conference
on
Algebraic Methodology and Software Technology

University of Twente

The Netherlands

Participants' Proceedings

APPLYING ALGEBRAIC LOGIC TO LOGIC

HAJNAL ANDRÉKA, ISTVÁN NÉMETI AND ILDIKÓ SAIN[*]
Mathematical Institute of the Hungarian Academy of Sciences
Budapest, P.O.B. 127, H-1364, Hungary

Abstract. Connections between Algebraic Logic and (ordinary) Logic. Algebraic counterpart of model theoretic semantics, algebraic counterpart of proof theory, and their connections. The class Alg(L) of algebras associated to any logic L. Equivalence theorems stating that L has a certain logical property iff Alg(L) has a certain algebraic property. (E.g. L admits a strongly complete Hilbert-style inference system iff Alg(L) is a finitely axiomatizable quasivariety. Similarly, L is compact iff Alg(L) is closed under taking ultraproducts; L has the Craig interpolation property iff Alg(L) has the amalgamation property, etc.)

Contents

1. Introduction

The idea of solving problems in logic by first translating them to algebra, then using the powerful methodology of algebra for solving them, and then translating the solution back to logic, goes back to Leibnitz and Pascal. Papers on the history of Logic (e.g. Anellis–Houser [**AH91**], Maddux [**Ma91**]) point out that this method was fruitfully applied in the 19th century not only to propositional logics but also to quantifier logics (De Morgan, Peirce, etc. applied it to quantifier logics too). The number of applications grew ever since. (Though some of these remained unnoticed, e.g. the celebrated Kripke–Lemmon completeness theorem for modal logic w.r.t. Kripke models was first proved by Jónsson and Tarski in 1948 using algebraic logic.)

For brevity, we will refer to the above method or procedure as "applying Algebraic Logic (AL) to Logic". This expression might be somewhat misleading since AL itself happens to be a part of logic, and we do not intend to deny this. We will use the expression all the same, and hope, the reader will not misunderstand our intention.

In items (i) and (ii) below we describe two of the main motivations for applying AL to Logic.

[*] We are very grateful to Ágnes Kurucz for substantial contribution to both the contents and the form of this work.

(i) This is the more obvious one: When working with a relatively new kind of problem, it is often proved to be useful to "transform" the problem into a well understood and streamlined area of mathematics, solve the problem there and translate the result back. Examples include the method of Laplace Transform in solving differential equations (a central tool in Electrical Engineering).

At this point we should dispell a misunderstanding: In certain circles of logicians there seems to be a belief that AL applies only to syntactical problems of logic and that semantical and model-theoretic problems are not treated by AL or at least not in their original model theoretic form. Nothing can be as far from the truth as this belief, as e.g. looking into the present work should reveal. A variant of this belief is that the main bulk of AL is about offering a cheap pseudo semantics to Logics as a substitute for intuitive, model theoretic semantics. Again, this is very far from being true. (This is a particularly harmful piece of misinformation, because, this "slander" is easy to believe if one looks only superficially into a few AL papers.) To illustrate how far this belief is from truth, the semantical-model theoretic parts of the present work emphasize that they start out from a logical system $\mathcal{L}$ whose semantics is as intuitive and as non-algebraic as it wants to be, and then we transform $\mathcal{L}$ into algebra, paying special attention to not distorting its semantics in the process; and anyway, finally we translate the solutions back to the very original non-algebraic framework (including model theoretical semantics).

In the present paper we define the algebraic counterpart $\mathsf{Alg}(\mathcal{L})$ of a logic $\mathcal{L}$ together with the algebraic counterpart $\mathsf{Alg}_2(\mathcal{L})$ of the semantical-model theoretical ingredients of $\mathcal{L}$. Then we prove equivalence theorems, which to essential logical properties of $\mathcal{L}$ associate natural and well investigated properties of $\mathsf{Alg}(\mathcal{L})$ such that if we want to decide whether $\mathcal{L}$ has a certain property, we will know what to ask from our algebraician colleague about $\mathsf{Alg}(\mathcal{L})$. The same devices are suitable for finding out what one has to change in $\mathcal{L}$ if we want to have a variant of $\mathcal{L}$ having a desirable property (which $\mathcal{L}$ lacks). To illustrate these applications we include several exercises (which deal with various concrete Logics). For all this, first we have to define what we understand by a logic $\mathcal{L}$ in general (because otherwise it is impossible to define e.g. the function Alg associating a class $\mathsf{Alg}(\mathcal{L})$ of algebras to each logic $\mathcal{L}$.

(ii) With the rapidly growing variety of applications of logic (in diverse areas like computer science, linguistics, AI, law, etc.) there is a growing number of new logics to be investigated. In this situation AL offers us a tool for economy and a tool for unification in various ways. One of these is that $\mathsf{Alg}(\mathcal{L})$ is always a class of algebras, therefore we can apply the same machinery namely Universal Algebra to study all the new logics. In other words we bring all the various logics to a kind of "normal form" where they can be studied by uniform methods. Moreover, for most choices of $\mathcal{L}$, $\mathsf{Alg}(\mathcal{L})$ tends to appear in the same "area" of Universal Algebra, hence specialized powerful methods lend themselves to studying $\mathcal{L}$. There is a fairly well understood "map" available for the landscape of Universal Algebra. By using our algebraization process and equivalence theorems we can project this "map" back to the (far less understood) landscape of possible logics.

2. General framework for studying logics

Defining a logic is an experience similar to defining a language. (This is no coincidence if you think about the applications of logic in e.g. theoretical linguistics.) So how do we define a language, say a programming language like Pascal. First one defines the *syntax* of Pascal. This amounts to defining the set of all Pascal programs. This definition tells us which strings of symbols count as Pascal programs and which do not. But this information in itself is not very useful, because having only this information enables the user to write programs but the user will have no idea what his programs will do. (This is more sensible if instead of Pascal we take a more esoteric language like ALGOL 68.) Indeed, the second, and more important step in defining Pascal amounts to describing what the various Pascal programs will do when executed. In other words we have to define the meaning, or *semantics* of the language, e.g. of Pascal. Defining semantics can be done in two steps, (i) we define the class M of *possible machines* that understand Pascal, and then (ii) to each machine $\mathfrak{M}$ and each string φ of symbols that counts as a Pascal program we tell what $\mathfrak{M}$ will do if we "ask" to execute φ. In other words we define the *meaning* $mean(\varphi, \mathfrak{M})$ of program φ in machine $\mathfrak{M}$.

The procedure remains basically the same if the language in question is not a programming language but something like a natural language or a simple declarative language like first–order logic. When teaching a foreign language e.g. German, one has to explain which strings of symbols are German sentences and which are not (e.g. "Der Tisch ist rot" is a German sentence while "Das Tisch ist rot" is not). This is called explaining the syntax of German. Besides this, one has to explain what the German sentences mean. This amounts to defining the semantics of German. If we want to formalize the definition of semantics (for, say, a fragment of German) then one again defines a class M of possible situations or in other words, "possible worlds" in which our German sentences are interpreted, and then to each situation $\mathfrak{M}$ and each sentence φ we define the meaning or denotation $mean(\varphi, \mathfrak{M})$ of φ in situation (or possible world) $\mathfrak{M}$.

At this point we could discuss the difference between a language and a logic, but we do not need that. It is enough to say that the two things are very–very similar.[1]

Soon (in Definition 2.1 below) we will define what we mean by a logic. Roughly speaking, a *logic* $\mathcal{L}$ is a triple

$$\mathcal{L} = \langle F_{\mathcal{L}}, M_{\mathcal{L}}, mean_{\mathcal{L}} \rangle ,$$

where

- $F_{\mathcal{L}}$ is a set, called the set of all *formulas* of $\mathcal{L}$,
- $M_{\mathcal{L}}$ is a class, called the class of all *models* (or *possible worlds*) of $\mathcal{L}$,
- $mean_{\mathcal{L}}$ is a function with domain $F_{\mathcal{L}} \times M_{\mathcal{L}}$, called the *meaning function* of $\mathcal{L}$.

[1] The philosophical minded reader might enjoy looking into the book [P89], cf. e.g. B.Partee's paper therein. More elementary ones are: Sain [S80/a] and [S80/b].

Intuitively, $F_{\mathcal{L}}$ is the collection of "texts" or "sentences" or "formulas" that can be "said" in the language $\mathcal{L}$. The meaning function tells us what the texts belonging to $F_{\mathcal{L}}$ mean in the possible worlds from $M_{\mathcal{L}}$.

Often, instead of $mean_{\mathcal{L}}$, we rather have a relation $\models_{\mathcal{L}} \subseteq M_{\mathcal{L}} \times F_{\mathcal{L}}$, called ***validity relation***. In more detail, very often from $mean_{\mathcal{L}}$ the relation $\models_{\mathcal{L}}$ is definable (and vica versa); but in general, we may have a logic $\mathcal{L}$ where $\models_{\mathcal{L}}$ does not make sense at all.

When no confusion is likely, we omit the subscripts $\mathcal{L}$ from $F_{\mathcal{L}}$, $M_{\mathcal{L}}$ etc.

A typical definition of F has the following recursive form. Two sets, P and LC are given; P is called the set of primitive or ***atomic*** formulas and LC is called the set of ***logical connectives*** (these are operation symbols with finite or infinite ranks). Then we require F be the smallest set H satisfying

(1) $P \subseteq H$, and

(2) for every $\varphi_1, \ldots, \varphi_n \in H$ and $f \in LC$ of rank n, $f(\varphi_1, \ldots, \varphi_n) \in H$.

For example, in propositional logic, if p_1, p_2 are propositional variables (atomic formulas according to our terminology), then $(p_1 \wedge p_2)$ is defined to be a formula (where $\wedge$ is a logical connective of rank 2).

For formulas $\varphi \in F$ and models $\mathfrak{M} \in M$, $mean(\varphi, \mathfrak{M})$ is defined in a uniform way (by some finite "schema").

For a logic $\mathcal{L} = \langle F, M, mean \rangle$, F belongs to the ***syntactic*** part, while M and $mean$ to the ***semanical*** part of $\mathcal{L}$. Figure 1. below illustrates the general pattern ("fan-structure") of a logic.

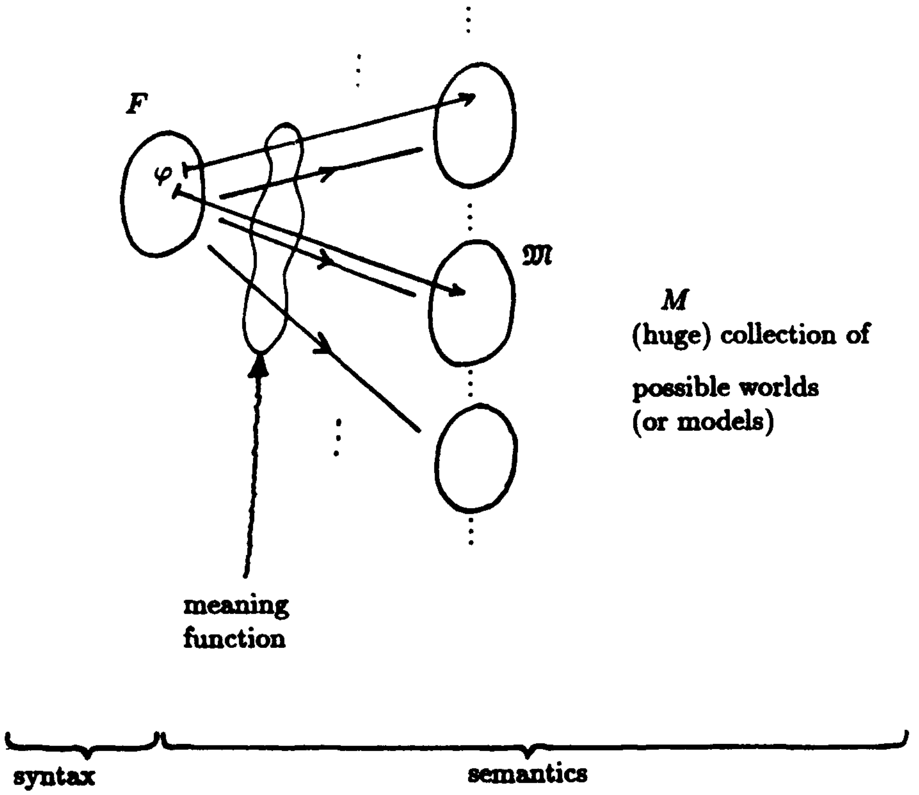


Figure 1

Though above we said that a logic only *roughly speaking* is a triple described above, in Definition 2.1 below we *call* such a triple a logic. This definition of a logic is very rude. However, we will see that it well serves the purposes of the present paper. Therefore we do not try here to give a more refined definition of a logic.

In Definition 2.1 we give the definition of a logic with validity relation $\models$. We will turn to logics with meaning functions only in section 2.1 later.

DEFINITION 2.1 (Logic):

By a *logic* $\mathcal{L}$ we mean an ordered triple

$$\mathcal{L} \overset{\text{def}}{=} \langle F_{\mathcal{L}}, M_{\mathcal{L}}, \models_{\mathcal{L}} \rangle,$$

where (i)–(iii) below hold.

(i) $F_{\mathcal{L}}$ (called the set of *formulas*) is a subset of finite sequences (called *words*) over some set X (called the *alphabet* of $\mathcal{L}$) that is,

$$F_{\mathcal{L}} \subseteq X^* \overset{\text{def}}{=} \{\langle a_0, \dots, a_{n-1}\rangle \ : \ n \in \omega,\ (\forall i < n)\ a_i \in X\};$$

(ii) $M_{\mathcal{L}}$ is a class (called the class of *models*);

(iii) $\models_{\mathcal{L}}$ (called the *validity relation*) is a relation between $M_{\mathcal{L}}$ and $F_{\mathcal{L}}$ that is, $\models_{\mathcal{L}} \subseteq M_{\mathcal{L}} \times F_{\mathcal{L}}$. ◄

DEFINITION 2.2 (Semantical Consequence):

Let $\mathcal{L} = \langle F_{\mathcal{L}}, M_{\mathcal{L}}, \models_{\mathcal{L}} \rangle$ be a logic. For every $\mathfrak{M} \in M_{\mathcal{L}}$, $\Sigma \subseteq F_{\mathcal{L}}$,

$$\mathfrak{M} \models_{\mathcal{L}} \Sigma \ \overset{\text{def}}{\iff} \ (\forall \varphi \in \Sigma)\mathfrak{M} \models_{\mathcal{L}} \varphi,$$
$$Mod_{\mathcal{L}}(\Sigma) \overset{\text{def}}{=} \{\mathfrak{M} \in M_{\mathcal{L}} \ : \ \mathfrak{M} \models_{\mathcal{L}} \Sigma\}.$$

A formula φ is said to be *valid*, in symbols $\models_{\mathcal{L}} \varphi$, iff $Mod_{\mathcal{L}}(\{\varphi\}) = M_{\mathcal{L}}$.

For any $\Sigma \cup \{\varphi\} \subseteq F_{\mathcal{L}}$,

$$\Sigma \models_{\mathcal{L}} \varphi \ \overset{\text{def}}{\iff} \ Mod(\Sigma) \subseteq Mod(\{\varphi\}).$$

If $\Sigma \models_{\mathcal{L}} \varphi$, then we say that φ is a *semantical consequence* of Σ (in logic $\mathcal{L}$). ◄

Now we define some basic logics. Though we think the reader is familiar with classical propositional logic, for fixing our notation, we start with the definition of it.

DEFINITION 2.3 (Propositional or Sentential Logic $\mathcal{L}_S$):

Let P be an arbitrary but fixed set, and let $\wedge$ a binary and $\neg$ a unary logical connective (operation symbol). P is called the set of all atomic formulas (or propositional variables) of propositional logic.

(1) The set F_S of formulas of propositional logic is defined to be the smallest set H satisfying the following two conditions:

- $P \subseteq H$, and
- $\varphi, \psi \in H \quad \Rightarrow \quad (\varphi \wedge \psi), (\neg\varphi) \in H$.

(2) The class M_S of *models* of propositional logic is defined as

$$M_S \overset{\text{def}}{=} \{\langle W, v\rangle : W \text{ is a non-empty set and } v : P \to \mathcal{P}(W)\}\,.$$

If $\mathfrak{M} = \langle W, v\rangle \in M_S$ then W is called the set of *possible worlds* (or *states* or *situations*) of $\mathfrak{M}$.

(3) Let $\langle W, v\rangle \in M_S$, $w \in W$, and $\varphi \in F_S$. We define the binary relation $w \Vdash_v \varphi$ by recursion on the complexity of φ as follows:

- if $p \in P$ then $\quad (w \Vdash_v \varphi \overset{\text{def}}{\iff} w \in v(p))$
- if $\psi_1, \psi_2 \in F_S$, then

$$\begin{aligned} w \Vdash_v \neg\psi_1 \quad &\overset{\text{def}}{\iff} \quad w \nVdash_v \psi_1 \\ w \Vdash_v \psi_1 \wedge \psi_2 \quad &\overset{\text{def}}{\iff} \quad w \Vdash_v \psi_1 \text{ and } w \Vdash_v \psi_2. \end{aligned}$$

If $w \Vdash_v \varphi$ then we say that φ is *true in* w, or w *forces* φ.

We say that φ is *true in* $\langle W, v\rangle$, in symbols $\langle W, v\rangle \models_S \varphi$ or $W \models_S \varphi[v]$, iff for every $w \in W$, $w \Vdash_v \varphi$.

Now, *propositional* (or *sentential*) *logic* is defined to be the triple

$$\mathcal{L}_S \overset{\text{def}}{=} \langle F_S, M_S, \models_S\rangle\,. \qquad ◄$$

EXERCISE 2.1:

Let $(\varphi \to \psi) \overset{\text{def}}{\iff} \neg(\varphi \wedge \neg\psi)$ and $\varphi \leftrightarrow \psi \overset{\text{def}}{\iff} ((\varphi \to \psi) \wedge (\psi \to \varphi))$. Prove that

- $\{\varphi\} \models_S \psi \iff \models_S (\varphi \to \psi)$
- $(\{\varphi\} \models_S \psi$ and $\{\psi\} \models_S \varphi) \iff \models_S (\varphi \leftrightarrow \psi)$. ◄

DEFINITION 2.4 (Modal logics $S5$, K; Arrow logics $\mathcal{L}_{\text{ARW0}}$, $\mathcal{L}_{\text{ARWRL}}$):

For each logic in this definition, first a relation $\Vdash$ similar to the one in Definition 2.3 will be given, and the validity relation $\models$ will be defined from $\Vdash$ exactly the same way as in in Definition 2.3.

The set of connectives of *modal logics* $S5$ and K is $\{\wedge, \neg, \Diamond\}$.

- The set of formulas (denoted as F_{S5}) of $S5$ is defined as that of $\mathcal{L}_S$ together with the following clause:

$$\varphi \in F_{S5} \implies \Diamond\varphi \in F_{S5}.$$

Let $M_{S5} \overset{\text{def}}{=} M_S$. The definition of $w \Vdash_v \varphi$ is the same as in the propositional case but we also have the case of $\Diamond$:

$$w \Vdash_v \Diamond\varphi \quad \overset{\text{def}}{\iff} \quad (\exists w' \in W)\; w' \Vdash_v \varphi.$$

Now, modal logic $S5$ is $S5 \overset{\text{def}}{=} \langle F_{S5}, M_{S5}, \models\rangle$.

- The formulas of logic K are those of $S5$: $F_K \overset{\text{def}}{=} F_{S5}$. The models of K are those of $S5$ together with a binary relation (called ***accessibility relation***) for each model. More precisely,

  $$M_K \overset{\text{def}}{=} \{\langle\langle W,v\rangle,R\rangle \ : \ \langle W,v\rangle \in M_S \text{ and } R \subseteq W \times W\}.$$

  The definition of $w \Vdash_v \varphi$ is as above, but in the case of $\Diamond$ we require that w' is accessible from w that is,

  $$w \Vdash_v \Diamond\varphi \overset{\text{def}}{\iff} (\exists w' \in W)(R(w,w') \text{ and } w' \Vdash_v \varphi).$$

  Then modal logic K is $K \overset{\text{def}}{=} \langle F_K, M_K, \models\rangle$.

The set of connectives of ***arrow logics*** $\mathcal{L}_{\text{ARWo}}$, $\mathcal{L}_{\text{ARWRL}}$ is $\{\wedge,\neg,\circ,\smile,Id\}$.

- The set of formulas of $\mathcal{L}_{\text{ARWo}}$ (denoted as F_{ARWo} and called ***arrow formulas***) is defined as follows. All sentential formulas are arrow formulas (i.e. $F_S \subseteq F_{ARWo}$), and

  $$\begin{aligned}&\varphi,\psi \in F_{\text{ARWo}} \implies \varphi\circ\psi,\ \varphi^{\smile} \in F_{\text{ARWo}}\\ &Id \in F_{\text{ARWo}}\end{aligned}$$

  The models are those of propositional logic $\mathcal{L}_S$ enriched with three accessibility relations. That is,

  $$\begin{aligned}M_{\text{ARWo}} \overset{\text{def}}{=} \{\langle\langle W,v\rangle,C_1,C_2,C_3\rangle \ : \ \langle W,v\rangle \in M_S,\ &C_1 \subseteq W\times W\times W,\\ &C_2 \subseteq W\times W,\ C_3 \subseteq W\}.\end{aligned}$$

  For sentential connectives $\neg$ and $\wedge$ the definition of $w \Vdash_v \varphi$ is the same as in the sentential case. For the new connectives we have:

  $$\begin{aligned}w \Vdash_v \varphi\circ\psi &\overset{\text{def}}{\iff} (\exists w_1,w_2 \in W)(C_1(w,w_1,w_2) \ \& \ w_1 \Vdash_v \varphi \ \& \ w_2 \Vdash_v \psi)\\ w \Vdash_v \varphi^{\smile} &\overset{\text{def}}{\iff} (\exists w' \in W)(C_2(w,w') \ \& \ w' \Vdash_v \varphi)\\ w \Vdash_v Id &\overset{\text{def}}{\iff} C_3(w).\end{aligned}$$

  Then arrow logic $\mathcal{L}_{\text{ARWo}}$ is $\mathcal{L}_{\text{ARWo}} \overset{\text{def}}{=} \langle F_{\text{ARWo}}, M_{\text{ARWo}}, \models\rangle$.

- The formulas of $\mathcal{L}_{\text{ARWRL}}$ are the arrow formulas, i.e. $F_{\text{ARWRL}} \overset{\text{def}}{=} F_{\text{ARWo}}$. The models are those of $\mathcal{L}_{\text{ARWo}}$ with the following restriction. For every model $\langle W,v\rangle \in M_{\text{ARWRL}}$, W is a binary relation on some set U that is, $W \subseteq U \times U$. Moreover, C_1 is relational composition, C_2 is the converse relation, and C_3 is the identity on U, respectively. More precisely, for any $w_1,w_2,w_3 \in W$, we let

  $$\begin{aligned}C_1(w_1,w_2,w_3) &\overset{\text{def}}{\iff} (\exists u_1,u_2,u_3 \in U)(w_1 = \langle u_1,u_3\rangle \ \&\\ &\qquad\qquad w_2 = \langle u_1,u_2\rangle \ \& \ w_3 = \langle u_2,u_3\rangle)\\ C_2(w_1,w_2) &\overset{\text{def}}{\iff} (\exists u_1,u_2 \in U)(w_1 = \langle u_1,u_2\rangle \ \& \ w_2 = \langle u_2,u_1\rangle)\\ C_3(w_1) &\overset{\text{def}}{\iff} (\exists u \in U)\ w_1 = \langle u,u\rangle.\end{aligned}$$

  Given these restrictions, the definition of $\Vdash$ is the same as in the previous case. Arrow logic $\mathcal{L}_{\text{ARWRL}}$ is $\mathcal{L}_{\text{ARWRL}} \overset{\text{def}}{=} \langle F_{\text{ARWRL}}, M_{\text{ARWRL}}, \models\rangle$. ◀

EXERCISE 2.2: Try to find similarities and differences between the logics $\mathcal{L}_S$, $S5$, K, $\mathcal{L}_{\text{ARW0}}$ and $\mathcal{L}_{\text{ARWRL}}$. ◄

EXERCISE 2.3: Consider the fragments $\mathcal{L}^0_{\text{ARW0}}$ and $\mathcal{L}^0_{\text{ARWRL}}$ of our arrow logics defined above which differ from the original versions only in that they do not contain the logical connectives $\smile$ and Id. Prove that $\mathcal{L}^0_{\text{ARW0}}$ is equivalent to $\mathcal{L}^0_{\text{ARWRL}}$ in the sense that they have the same semantical consequence relation $\models$.

Prove that $\mathcal{L}_{\text{ARW0}}$ is not equivalent, in the above sense, to $\mathcal{L}_{\text{ARWRL}}$. ◄

DEFINITION 2.5 (First-order Logic with n variables $\mathcal{L}_n$):
First-order logic with n variables is defined to be a triple

$$\mathcal{L}_n \stackrel{\text{def}}{=} \langle F_n, M_n, \models_n \rangle,$$

for which conditions (1)–(3) below hold.

(1) Let $V \stackrel{\text{def}}{=} \{v_0, \dots, v_{n-1}\}$ be a set, called the set of *variables*. Let the set P of *atomic formulas* be defined as $P \stackrel{\text{def}}{=} \{r_i(v_0 \dots v_{n-1}) : i \in I\}$ for some set I. Then the set F_n of formulas is the smallest set H satisfying
- $P \subseteq H$
- $v = w \in H$ for each $v, w \in V$
- $\varphi, \psi \in H,\ v \in V \implies \varphi \wedge \psi,\ \neg\varphi,\ \exists v\varphi \in H.$

(2) The class M_n of *models* of $\mathcal{L}_n$ is defined by

$$M_n \stackrel{\text{def}}{=} \{\langle A, R_i \rangle_{i \in I} \ : \ A \text{ is a non-empty set and } R_i \subseteq {}^nA\ (i \in I)\}.$$

If $\mathfrak{M} = \langle A, R_i \rangle_{i \in I} \in M_n$ then A is called the *universe* (or *carrier*) of $\mathfrak{M}$.

(3) Let $\mathfrak{M} = \langle A, R_i \rangle_{i \in I} \in M_n$, $q \in {}^nA$, and $\varphi \in F_n$. We define the ternary relation $\mathfrak{M} \models \varphi[q]$ by induction on the complexity of φ:
- $\mathfrak{M} \models r_i(v_0 \dots v_{n-1})[q] \stackrel{\text{def}}{\iff} q \in R_i \quad (i \in I)$
- $\mathfrak{M} \models v_i = v_j[q] \stackrel{\text{def}}{\iff} q_i = q_j \quad (i, j \in I)$
- if $\psi_1, \psi_2 \in F_S$, then

$$\mathfrak{M} \models \neg\psi_1[q] \stackrel{\text{def}}{\iff} \text{not } \mathfrak{M} \models \psi_1[q]$$

$$\mathfrak{M} \models \psi_1 \wedge \psi_2[q] \stackrel{\text{def}}{\iff} \mathfrak{M} \models \psi_1[q] \text{ and } \mathfrak{M} \models \psi_2[q]$$

$$\mathfrak{M} \models \exists v_i \psi_1[q] \stackrel{\text{def}}{\iff} (\exists q' \in {}^nA)[(\forall i \neq j \in I)\ q'_j = q_j \ \& \ \mathfrak{M} \models \psi_1[q']].$$

If $\mathfrak{M} \models \varphi[q]$ then we say that the *evaluation q satisfies φ in model $\mathfrak{M}$*.

We say that $\mathfrak{M} \models_n \varphi$ iff for every $q \in {}^nA$ $\mathfrak{M} \models \varphi[q]$. ◄

* * *

DEFINITION 2.6 (Theory):
Let $\mathcal{L} = \langle F, M, \models \rangle$ be any logic. For any $K \subseteq M$ let the *theory of* K be defined as

$$Th(K) \overset{\text{def}}{=} \{\varphi \in F \ : \ (\forall \mathfrak{M} \in K)\ \mathfrak{M} \models \varphi\}. \qquad ◄$$

Recall the notions of recursively enumerable and decidable sets (of formulas).

DEFINITION 2.7 (Decidability of Logics):
We say that a logic is *decidable* iff $Th(M_{\mathcal{L}})$ is a decidable set of formulas. ◄

THEOREM 2.2 (Decidability of $\mathcal{L}_S$ and $S5$). *Propositional logic $\mathcal{L}_S$ and modal logic $S5$ are decidable.* ▮

This theorem will be proved later. We will show that, in both cases, the set of valid formulas is recursively enumerable (r.e.) and that these logics have the finite model property (to be defined later).

2.1. Logics with satisfaction and/or meaning

Defining a logic as $\langle F, M, \models \rangle$ with $\models \subseteq M \times F$ is an oversimplification for the following reasons. If we look at the logics in Defs.2.3–5 ($S5$, $\mathcal{L}_n$, etc.) we will notice that they contain a richer semantical structure than just a binary relation $\models \subseteq M \times F$. In each case, there is a class Par of parameters and a *ternary* relation $\Vdash \subseteq M \times Par \times F$ which is usually called the *satisfaction relation.* In the definition of $S5$ we should have written $\langle \mathfrak{M}, w \rangle \Vdash \varphi$ or at least $\mathfrak{M}, w \Vdash \varphi$ instead of $w \Vdash_v \varphi$. However, for simplicity we used the latter, and we used the subscript "v" to indicate the presence of $\mathfrak{M}$. Anyway, a little reflection reveals that the definition of $S5$ uses a ternary relation "$\mathfrak{M}, w \Vdash \varphi$", where w was called a possible situation (world) of $\mathfrak{M}$.

The same applies to $\mathcal{L}_{\mathrm{ARW}0}$, etc. Perhaps the least trivial case is that of $\mathcal{L}_n$. There we use "$\mathfrak{M}, q \Vdash \varphi$", where $\mathfrak{M} = \langle A, R_i : i \in I \rangle$ and $q \in {}^nA$. In case of $\mathcal{L}_n$, the traditional way of writing "$\mathfrak{M}, q \Vdash \varphi$" is "$\mathfrak{M} \models \varphi[q]$" and is pronounced as the evaluation q satisfies φ in the model $\mathfrak{M}$.

In each of the logics we saw so far, first $\langle F, M, \Vdash \rangle$ is defined and then the binary *validity relation* $\models$ is derived (in some way) from the deeper, more substantial relation $\Vdash$ (in a sense, $\models$ was always a "simplified part" of $\Vdash$). In all of our examples the following derivation of $\models$ from $\Vdash$ works.

$$(*) \qquad \mathfrak{M} \models \varphi \quad \text{iff} \quad (\forall \psi \in F)\forall w\big[(\mathfrak{M}, w \Vdash \psi) \Longrightarrow (\mathfrak{M}, w \Vdash \varphi)\big].$$

EXERCISE 2.1.1: Check that $(*)$ above is true for all the logics we defined so far. ◀

EXERCISE 2.1.2: Show that while $\models$ can be derived (i.e. recovered) from $\Vdash$, in most of our logics $\Vdash$ cannot be recovered from $\models$. ◀

There are ways other than $(*)$ above for deriving $\models$ from $\Vdash$. E.g. $(**)$ below works too (for the logics considered so far):

$$(**) \qquad \mathfrak{M} \models \varphi \quad \text{iff} \quad \forall w\big[(\mathfrak{M}, w \Vdash (\varphi \leftrightarrow \varphi)) \Longrightarrow (\mathfrak{M}, w \Vdash \varphi)\big]$$

(cf. Exercise 2.1).

DEFINITION 2.1.1 (Logic with Satisfaction):
By a ***logic with satisfaction*** we understand a quadruple $\mathcal{L} = \langle F, M, \models, \Vdash\rangle$, where

(i) $\langle F, M, \models\rangle$ is a logic in the sense of Def.2.1;
(ii) $\models$ is derived from $\Vdash$ in a manner similar to $(*)$ or $(**)$ above. ◀

We know that item (ii) in the above definition is somewhat vague. If that would disturb the reader, it is safe to substitute $(*)$ for (ii).[2]

EXERCISE 2.1.3: Show logics (in the sense of Def.2.1) in which though $\models$ can be derived fr.m $\Vdash$ in some way, neither $(*)$ nor $(**)$ hold. ◀

Instead of the above concept of a logic with satisfaction, we will use a less ad-hoc variant which is at least as general as the above one. The idea is the following. Given a syntactic entity (a formula) $\varphi \in F$, and a possible world $\mathfrak{M} \in M$, instead of giving a truth value ($\mathfrak{M} \models \varphi$ or $\mathfrak{M} \not\models \varphi$) to φ in $\mathfrak{M}$, we associate a *meaning* to φ in $\mathfrak{M}$. Certainly, the most natural (and most general) thing a syntactic expression φ might have in an environment or world $\mathfrak{M}$ is a meaning. What that meaning will be might depend on the kind of expression φ we are looking at, and the kind of $\mathfrak{M}$ we are having in mind. E.g. the meaning might be a truth value (*True*, *False*); or an element of the set $\mathfrak{M}$ denoted by φ; more generally a denotation; if φ is a program and $\mathfrak{M}$ is a machine then the meaning of φ might be the function computed by φ in $\mathfrak{M}$; or if we are in a logic with satisfaction then it might be the set $\{w : \mathfrak{M}, w \Vdash \varphi\}$.

DEFINITION 2.1.2 (Logic with Meaning):
By a ***logic with meaning*** we understand a quadruple $\mathcal{L} = \langle F, M, \models, \mathit{mean}\rangle$, where *mean* is a function with domain $F \times M$ and conditions (i), (ii) below hold.

(i) $\langle F, M \models\rangle$ is a logic in the sense of Def.2.1;
(ii) $\models$ is derived from *mean* either by $(***)$ below or by a similar definition.

$$(***) \qquad \mathfrak{M} \models \varphi \quad \text{iff} \quad (\forall \psi \in F)\big[\mathit{mean}(\psi, \mathfrak{M}) \subseteq \mathit{mean}(\varphi, \mathfrak{M})\big],$$

for all $\varphi \in F$, $\mathfrak{M} \in M$. ◀

[2] We wanted to keep our definition more general but that is not essential for the present work. Also we felt that while condition $(*)$ is not so essential to the concept of a logic as the admittedly vague formulation of (ii).

A similar remark applies to (ii) above as the one below Def.2.1.1.

EXERCISE 2.1.4: Prove that logics with satisfaction and those with meaning are equivalent in the following sense:

(1) To every logic $\mathcal{L}_{mean} = \langle F, M, \models, mean \rangle$ with meaning there is some logic $\mathcal{L}_{sat} = \langle F, M, \models, \Vdash \rangle$ with satisfaction such that they are interdefinable; and

(2) To every $\mathcal{L}_{sat}$ there is an $\mathcal{L}_{mean}$ as in (1) above.

(Hint: Assume $\mathcal{L}_{mean} = \langle \dots mean \rangle$ is given. Let

$$Par \overset{\text{def}}{=} \bigcup Rng(mean) = \bigcup \{ mean(\varphi, \mathfrak{M}) \ : \ \varphi \in F, \ \mathfrak{M} \in M \}.$$

For $w \in Par$ define $\left[(\mathfrak{M}, w \Vdash \varphi) \overset{\text{def}}{\Longleftrightarrow} w \in mean(\varphi, \mathfrak{M}) \right]$. In the other direction (i.e. assuming $\mathcal{L}_{sat}$ is given first) $mean(\varphi, \mathfrak{M}) \overset{\text{def}}{=} \{ w : \mathfrak{M}, w \Vdash \varphi \}$. Show that using these definitions $\mathcal{L}_{mean}$ and $\mathcal{L}_{sat}$ are completely recoverable from each other.) ◀

EXERCISE 2.1.5: Show logics (in the sense of Def.2.1) in which though there is a "sensible" meaning function, condition $(***)$ above does not hold. (Hint: Try e.g. many–valued logics.) ◀

3. Bridge between the world of logics and the world of algebras

The algebraic counterpart of classical sentential logic $\mathcal{L}_S$ is the variety BA of Boolean algebras. Why is this so important? The answer lies in the general experience that it is usually much easier to solve a problem concerning $\mathcal{L}_S$ by translating it to BA, solving the algebraic problem, and then translating the result back to $\mathcal{L}_S$ (then solving it directly in $\mathcal{L}_S$).

In this section we extend applicability of BA to $\mathcal{L}_S$ to applicability of algebra in general to logics in general. We will introduce a standard translation method from logic to algebra, which to each logic $\mathcal{L}$ associates a class of algebras $\mathsf{Alg}_1(\mathcal{L})$. (Of course, $\mathsf{Alg}_1(\mathcal{L}_S)$ will be BA.) Further, this translation method will tell us how to find the algebraic question corresponding to a logical question. If the logical question is about $\mathcal{L}$ then its algebraic equivalent will be about $\mathsf{Alg}_1(\mathcal{L})$. For example, if we want to decide whether $\mathcal{L}$ has the proof theoretic property called Craig's interpolation property, then it is sufficient to decide whether $\mathsf{Alg}_1(\mathcal{L})$ has the so called amalgamation property (for which there are powerful methods in the literature of algebra). If the logical question concerns connections between several logics, say between $\mathcal{L}_1$ and $\mathcal{L}_2$, then the algebraic question will be about connections between $\mathsf{Alg}_1(\mathcal{L}_1)$ and $\mathsf{Alg}_1(\mathcal{L}_2)$. (The latter are quite often simpler, hence easier to investigate.)

3.1. Basic concepts

The definition of logic in section 2 is very wide. Actually, it is too wide for proving interesting theorems about logics. Now we will define a subclass of logics which we will call nice logics. Our notion of nice logic is wide enough to cover the logics mentioned in the previous section, moreover, it is broad enough to cover almost all logics investigated in the literature. (Certain quantifier logics might need a little reformulation for this, but that reformulation does not effect the essential aspects of the logic in question as we will see.) On the other hand, the class of nice logics is narrow enough for proving interesting theorems about them, i.e., we will be able to establish typical logical facts that hold for most logics studied in the literature.

Before reading Def.3.1.1 below, it might be useful to contemplate the common features of the logics studied so far, e.g. $\mathcal{L}_S$, $S5$, $\mathcal{L}_{\mathrm{ARWo}}$, $\mathcal{L}_n$. When presenting this material in class, many more logics were discussed in order to motivate the definition of a nice logic. Some of these logics are collected in section 3.4 below. It might be a good idea to look into 3.4 too before reading the definition below.

In all the logics studied so far (and also in 3.4), the biconditional "$\leftrightarrow$" is available as a derived connective. In condition (3) of Def.3.1.1 there will occur a new symbol "∇" denoting a derived connective of the logic in question. At first reading it is a good idea to identify "∇" with our old biconditional "$\leftrightarrow$". Certainly, if we replaced condition (3) with the simpler assumption that "$\leftrightarrow$" is expressible in our logic $\mathcal{L}$ then all theorems would remain true. However, at a second reading of the definition it might be useful to observe that our condition (3) is a weaker assumption then expressibility of "$\leftrightarrow$" (and that this makes the class of nice logics broader).

We also note that the theorems of section 3.2 below (based on the next definition) can be proved in a more general setting (cf. [ANS84]). Here we do restrictions in order to make the methodology more transparent.

DEFINITION 3.1.1 (Nice Logic, Strongly Nice Logic):
Let $\langle F, M, \models\rangle$ be a logic in the sense of Definition 2.1 (i.e. F is a set, M is a class, and $\models \subseteq M \times F$.

We say that $\mathcal{L}$ is a *nice logic* if conditions (1–4) below hold for $\mathcal{L}$.

(1) A finite set $Cn(\mathcal{L})$, called the set of ***logical connectives*** of $\mathcal{L}$, is fixed. Every $c \in Cn(\mathcal{L})$ has some rank $rank(c) \in \omega$. The set of all logical connectives of rank k is denoted by $Cn_k(\mathcal{L})$.

There is a set P, called the set of ***atomic formulas*** (or ***parameters*** or ***propositional variables*** or ...), such that F is the smallest set satisfying conditions (a–b) below.

(a) $P \subseteq F$,

(b) if $c \in Cn_k(\mathcal{L})$ and $\varphi_1, \ldots, \varphi_k \in F$ then $c(\varphi_1, \ldots, \varphi_k) \in F$.

The word–algebra generated by P using the logical connectives from $Cn(\mathcal{L})$ as algebraic operations is denoted by $\underset{\sim}{F}$ that is, $\underset{\sim}{F} = \langle F, c\rangle_{c \in Cn(\mathcal{L})}$. $\underset{\sim}{F}$ is called the ***formula algebra*** of $\mathcal{L}$.

(2) We assume that a function ***mean*** is given with $Dom(mean) = F \times M$ and $mean_{\mathfrak{M}} \overset{\text{def}}{=} \langle mean(\varphi, \mathfrak{M}) : \varphi \in F\rangle$ is a homomorphism from $\underset{\sim}{F}$ for every $\mathfrak{M}$ (cf. section 2.1).

(3) We assume that there are ***"derived" connectives*** "*True*" (zero–ary) and "∇" (binary) of $\mathcal{L}$ with the following properties:

(i) $(\forall \mathfrak{M} \in M)(\forall \varphi, \psi \in F)\big[\mathfrak{M} \models (\varphi \nabla \psi) \iff mean_{\mathfrak{M}}(\varphi) = mean_{\mathfrak{M}}(\psi)\big]$.

(ii) $(\forall \mathfrak{M} \in M)(\forall \varphi \in F)\big[\mathfrak{M} \models True \nabla \varphi \iff \mathfrak{M} \models \varphi\big]$.

(By "derived" we mean that "*True*" and "∇" are not necessarily members of $Cn(\mathcal{L})$. They are only "built up" from elements of $Cn(\mathcal{L})$. But we do not know from which elements of $Cn(\mathcal{L})$ "*True*" or "∇" are built up, or how. We do not care!)

(4) $(\forall \psi, \varphi_0, \ldots, \varphi_n \in F)(\forall p_0, \ldots, p_n \in P)\big[\models \psi(\overline{p}) \implies \models \psi(\overline{p}/\overline{\varphi})\big]$,
where $\overline{p} = \langle p_0, \ldots, p_n\rangle$, $\overline{\varphi} = \langle \varphi_0, \ldots, \varphi_n\rangle$, and $\psi(\overline{p}/\overline{\varphi})$ denotes the formula that we get from ψ after substituting φ_i for every occurrence of p_i $(0 \leq i \leq n)$ in ψ. We refer to this condition as '***$\mathcal{L}$ has the substitution property***'.

$\mathcal{L}$ is called ***strongly nice*** iff it is nice and satisfies condition (5) below.

(5) $(\forall s \in {}^{P}F)(\forall \mathfrak{M} \in M)(\exists \mathfrak{N} \in M)(\forall \varphi(p_{i_0}, \ldots, p_{i_n}) \in F)$

$$(+) \qquad mean_{\mathfrak{N}}(\varphi) = mean_{\mathfrak{M}}(\varphi(p_{i_0}/s(p_{i_0}), \ldots, p_{i_n}/s(p_{i_n})))\,.$$

Let $\hat{s} \in {}^{F}F$ be the natural extension of s to $\underset{\sim}{F}$. Then (+) says $mean_{\mathfrak{N}}(\varphi) = mean_{\mathfrak{M}}(\hat{s}(\varphi))$. If this property holds, then we say that the logic '*$\mathcal{L}$ has the semantical substitution property*' (the model $\mathfrak{N}$ is the substituted version of $\mathfrak{M}$ along substitution s). ◀

Recall that if $\mathfrak{A}$ and $\mathfrak{B}$ are two similar algebras, then $Hom(\mathfrak{A}, \mathfrak{B})$ denotes the set of all homomorphisms from $\mathfrak{A}$ into $\mathfrak{B}$.

REMARKS 3.1.1.1:

(i) An equivalent form of (+) above is the very natural condition

$$\Big(\forall h \in Hom(\underset{\sim}{F}, \underset{\sim}{F})\Big)\,(\forall \mathfrak{M} \in M)(\exists \mathfrak{N} \in M)\ mean_{\mathfrak{N}} = mean_{\mathfrak{M}} \circ h\,.$$

Since h is just a substitution, this form makes it explicit that $\mathfrak{N}$ is the h–substituted version of $\mathfrak{M}$. Other equivalent version is the following.

$$(\forall \mathfrak{M} \in M)(\forall h \in Hom(\underset{\sim}{F}, mean_{\mathfrak{M}}(\underset{\sim}{F})))(\exists \mathfrak{N} \in M)\ mean_{\mathfrak{N}} = h.$$

(ii) Item (2) of Definition 3.1.1 above is a purely logical criterion. Namely, it is Frége's principle of compositionality.

(iii) Item (3)(i) and (ii) of Definition 3.1.1 above give the following connection between $\models$ and *mean*:

$$(\forall \varphi \in F)\big[\models \varphi \implies (\forall \mathfrak{M} \in M)\ mean_{\mathfrak{M}}(\varphi) = mean_{\mathfrak{M}}(True)\big].$$

(iv) In the presence of (3) of Definition 3.1.1 above, semantical substitution property ((5)) implies substitution property ((4)). ◀

EXERCISE 3.1.1: Show that $\mathcal{L}_n$, $\mathcal{L}_S$, $S5$, $\mathcal{L}_{\mathrm{ARW0}}$ and $\mathcal{L}_{\mathrm{ARWRL}}$ are strongly nice logics. (Hint: In each case, "$\leftrightarrow$" is good for "∇".) ◀

EXERCISE 3.1.2: Show logics where "∇" is not our old biconditional "$\leftrightarrow$". (E.g., in $S5$ we can also take $\Box(\Phi_1 \leftrightarrow \Phi_2)$ as $\Phi_1 \nabla \Phi_2$.) ◀

DEFINITION 3.1.2: (Algebraic Counterpart of a Logic)

Let $\mathcal{L} = \langle F, M, \models \rangle$ be a logic satisfying conditions (1),(2) of Definition 3.1.1 above.

(i) Let $K \subseteq M$. Then for every $\varphi, \psi \in F$

$$\varphi \sim_K \psi \overset{\text{def}}{\iff} (\forall \mathfrak{M} \in K)\ mean_{\mathfrak{M}}(\varphi) = mean_{\mathfrak{M}}(\psi).$$

$$\mathsf{Alg}_1(\mathcal{L}) \overset{\text{def}}{=} \mathbf{I}\left\{\underset{\sim}{F}/\sim_K\ :\ K \subseteq M\right\}.$$

(ii)

$$\mathsf{Alg}_2(\mathcal{L}) \overset{\text{def}}{=} \mathbf{I}\left\{mean_{\mathfrak{M}}(\underset{\sim}{F})\ :\ \mathfrak{M} \in M\right\},$$

where $mean_{\mathfrak{M}}$ was defined in item (2) of Definition 3.1.1, and for any homomorphism $h : \mathfrak{A} \longrightarrow \mathfrak{B}$, $h(\mathfrak{A})$ is the homomorphic image of $\mathfrak{A}$ along h i.e., $h(\mathfrak{A})$ is the smallest subalgebra of $\mathfrak{B}$ such that $h : \mathfrak{A} \longrightarrow h(\mathfrak{A})$. ◀

FACT 3.1.2.1: For nice logics

$$\mathsf{Alg}_1(\mathcal{L}) = \mathbf{I}\left\{\underset{\sim}{F}/\sim_\Gamma \,:\, \Gamma \subseteq F\right\},$$

where $\varphi \sim_\Gamma \psi \overset{\text{def}}{\iff} (\forall \mathfrak{M} \in M)(\mathfrak{M} \models \Gamma \Longrightarrow mean_{\mathfrak{M}}(\varphi) = mean_{\mathfrak{M}}(\psi))$.

PROOF: For every $K \subseteq M$ $\underset{\sim}{F}/\sim_K = \underset{\sim}{F}/\sim_{Th(K)}$, and for every $\Gamma \subseteq F$ $\underset{\sim}{F}/\sim_\Gamma = \underset{\sim}{F}/\sim_{Mod(\Gamma)}$ hold (cf. Definitions 2.2 and 2.6). ◀

EXERCISE 3.1.3: Show that for any logic $\mathcal{L}$ satisfying conditions (1),(2) of Definition 3.1.1

- $\mathsf{Alg}_2(\mathcal{L}) \subseteq \mathsf{Alg}_1(\mathcal{L})$
- $\mathbf{SP}\mathsf{Alg}_1(\mathcal{L}) = \mathbf{SP}\mathsf{Alg}_2(\mathcal{L})$. ◀

Recall the definitions of the class BA of Boolean algebras and of the class Cs_1 of one–dimensional cylindric set algebras.

EXERCISE 3.1.4: Prove that

(i) $\mathsf{Alg}_2(\mathcal{L}_S) = \mathsf{BA}$
(ii) $\mathsf{Alg}_2(S5) = \mathsf{Cs}_1$. ◀

The class RRA of representable relation algebras and its relativized version will be introduced and investigated in Chapter I There we will see that $\mathsf{Alg}_2(\mathcal{L}_{\mathrm{ARWRL}})$ coincides with the relativized version of RRA.

* * *

Next we turn to *inference systems*. Inference systems (usually denoted as $\vdash$) are syntactical devices serving to recapture (or at least to approximate) the semantical consequence relation $\models_\mathcal{L}$ of the logic $\mathcal{L}$. The idea is the following. Suppose $\Sigma \models_\mathcal{L} \varphi$. This means that, in the logic $\mathcal{L}$, the assumptions collected in Σ semantically imply the conclusion φ. (In any possible world $\mathfrak{M}$ of $\mathcal{L}$ that is, in any $\mathfrak{M} \in M_\mathcal{L}$, whenever Σ is valid in $\mathfrak{M}$, then also φ is valid in $\mathfrak{M}$.) Then we would like to be able to reproduce this relationship between Σ and φ by purely syntactical, "finitistic" means. That is, by applying some formal rules of inference (and some axioms of the logic $\mathcal{L}$) we would like to be able to derive φ from Σ by using "paper and pencil" only. In particular, such a derivation will always be a finite string of symbols. If we can do this, that will be denoted by $\Sigma \vdash \varphi$.

DEFINITION 3.1.3 (Formula Scheme):

Let $\mathcal{L}$ be a nice logic with the finite set $Cn(\mathcal{L})$ of logical connectives (cf. (1) of Def.3.1.1). Fix a countable set $A = \{A_i : i < \omega\}$, called the set of *formula variables*.

The set $Fms_{\mathcal{L}}$ of *formula schemes* of $\mathcal{L}$ is the smallest set satisfying conditions (a–b) below.

(a) $A \subseteq Fms_{\mathcal{L}}$,

(b) if $c \in Cn_k(\mathcal{L})$ and $\Phi_1, \ldots, \Phi_k \in Fms_{\mathcal{L}}$ then $c(\Phi_1, \ldots, \Phi_k) \in Fms_{\mathcal{L}}$.

An *instance of a formula scheme* is given by substituting formulas for the formula variables in it. ◄

DEFINITION 3.1.4 (Hilbert–style Inference System):

Let $\mathcal{L}$ be a nice logic. An *inference rule* of $\mathcal{L}$ is a pair $\langle\langle B_1, \ldots, B_n\rangle, B_0\rangle$, where every B_i $(i \leq n)$ is a formula scheme. This inference rule will be denoted by

$$\frac{B_1, \ldots, B_n}{B_0}.$$

An *instance of an inference rule* is given by substituting formulas for the formula variables in the formula schemes occurring in the rule.

A *Hilbert–style inference system* (or *calculus*) for $\mathcal{L}$ is a finite set of formula schemes (called *axiom schemes*) together with a finite set of inference rules. ◄

DEFINITION 3.1.5 (Derivability):

Let $\mathcal{L}$ be a nice logic and let $\vdash$ be a Hilbert–style inference system for $\mathcal{L}$. Assume $\Sigma \cup \{\varphi\} \subseteq F_{\mathcal{L}}$. We say that φ is $\vdash$*–derivable* (or *provable*) from Σ iff there is a finite sequence $\langle\varphi_1, \ldots, \varphi_n\rangle$ of formulas (an $\vdash$*-proof of* φ *from* Σ) such that φ_n is φ and for every $1 \leq i \leq n$

- $\varphi_i \in \Sigma$ or
- φ_i is an instance of an axiom scheme (an *axiom* for short) of $\vdash$ or
- there are $j_1, \ldots, j_k < i$, and there is an inference rule of $\vdash$ such that $\frac{\varphi_{j_1}, \ldots, \varphi_{j_k}}{\varphi_i}$ is an instance of this rule.

We write $\Sigma \vdash \varphi$ if φ is $\vdash$–provable from Σ. (We will often identify an inference system $\vdash$ with the corresponding derivability relation.) ◄

DEFINITION 3.1.6 (Complete and Sound Hilbert-type Inference System):

Let $\mathcal{L}$ be a nice logic and let $\vdash$ be a Hilbert–type inference system for $\mathcal{L}$. Then

- $\vdash$ is *weakly complete* for $\mathcal{L}$ iff
$$(\forall\varphi \in F_{\mathcal{L}}) \models_{\mathcal{L}} \varphi \implies \vdash \varphi;$$
- $\vdash$ is *finitely complete* for $\mathcal{L}$ iff
$$(\forall\Sigma \subseteq_{\omega} F_{\mathcal{L}})(\forall\varphi \in F_{\mathcal{L}})\ \Sigma \models_{\mathcal{L}} \varphi \implies \Sigma \vdash \varphi$$
that is, we consider only finite Σ's;
- $\vdash$ is *strongly complete* for $\mathcal{L}$ iff
$$(\forall\Sigma \subseteq F_{\mathcal{L}})(\forall\varphi \in F_{\mathcal{L}})\ \Sigma \models_{\mathcal{L}} \varphi \implies \Sigma \vdash \varphi;$$
- $\vdash$ is *weakly sound* for $\mathcal{L}$ iff
$$(\forall\varphi \in F_{\mathcal{L}}) \vdash \varphi \implies \models_{\mathcal{L}} \varphi;$$
- $\vdash$ is *strongly sound* for $\mathcal{L}$ iff
$$(\forall\Sigma \subseteq F_{\mathcal{L}})(\forall\varphi \in F_{\mathcal{L}})\ \Sigma \vdash \varphi \implies \Sigma \models_{\mathcal{L}} \varphi. \quad ◄$$

THEOREM 3.1.1 (Strong Completeness of $\mathcal{L}_S$ and $S5$). *There are strongly complete and strongly sound Hilbert-type inference systems for $\mathcal{L}_S$ and for $S5$.*

We will prove this theorem in section 3.2 below, using methods of Universal Algebraic Logic. ▮

We will also prove in section 3.2 that the arrow logics introduced in Def.2.4 also admit strongly complete and strongly sound Hilbert-tyoe inference systems.

3.2. Main theorems

In Theorem 3.2.1 below, we will give a sufficeut and necessary condition for a strongly nice logic to have a finitely complete Hilbert-style inference system.

THEOREM 3.2.1. *Assume $\mathcal{L}$ is strongly nice. Then*

$$\mathsf{Alg}_2(\mathcal{L}) \textit{ generates a finitely axiomatizable quasivariety}$$
$$\Longleftrightarrow$$
$$(\exists \textit{ Hilbert-style } \vdash)(\vdash \textit{ is finitely complete and strongly sound for } \mathcal{L}).$$

Proof of $(\Longrightarrow)$: Let $\Phi_0, \Phi_1, \ldots$ denote formula variables, $\tau_0, \tau_1, \ldots$ denote formula schemes, $\bar{\Phi}$ denote sequence of formula variables and $\bar{x}$ denote sequence of variables. Assume that Ax is a finite set of quasiequations axiomatizing the quasivariety generated by $\mathsf{Alg}_2(\mathcal{L})$ and define a Hilbert-style inference system $\vdash_{Ax}$ as follows:
AXIOM SCHEME: $(\Phi_0 \nabla \Phi_0)$ *(reflexivity).*
INFERENCE RULES: If $(\tau_1(\bar{x}) = \tau_1'(\bar{x})\ \&\ \cdots\ \&\ \tau_n(\bar{x}) = \tau_n'(\bar{x})) \Rightarrow \tau_0(\bar{x}) = \tau_0'(\bar{x}) \in Ax$, then

$$\frac{\tau_1(\bar{\Phi})\nabla\tau_1'(\bar{\Phi}), \ldots, \tau_n(\bar{\Phi})\nabla\tau_n'(\bar{\Phi})}{\tau_0(\bar{\Phi})\nabla\tau_0'(\bar{\Phi})}$$

is a rule. Other rules are:

$$\frac{\Phi_0\nabla\Phi_1,\ \Phi_1\nabla\Phi_2}{\Phi_0\nabla\Phi_2} \qquad \textit{(transitivity)},$$

$$\frac{\Phi_0\nabla\Phi_1}{\Phi_1\nabla\Phi_0} \qquad \textit{(symmetry)},$$

$$(\forall c \in Cn_k(\mathcal{L})) \quad \frac{\Phi_1\nabla\Phi_1', \ldots, \Phi_k\nabla\Phi_k'}{c(\Phi_1, \ldots, \Phi_k)\nabla c(\Phi_1', \ldots, \Phi_k')} \qquad \textit{(congruence)},$$

$$\frac{\Phi_0\nabla\, True}{\Phi_0}, \qquad \frac{\Phi_0}{\Phi_0\nabla\, True}.$$

We will show that the inference system $\vdash_{Ax}$ is finitely complete and strongly sound for $\mathcal{L}$.

For any set Σ of formulas we define $\psi \sim_\Sigma \psi' \overset{\text{def}}{\Longleftrightarrow} \Sigma \vdash_{Ax} (\psi \nabla \psi')$. Note that, by the definition of $\vdash_{Ax}$ and by Definition 3.1.5, $\sim_\Sigma$ is a congruence relation on $\underset{\sim}{F}$ for any Σ.

Claim 3.2.1.1: $(\underset{\sim}{F}/{\sim_\Sigma}) \models Ax$.

PROOF OF CLAIM 3.2.1.1: Let $q \in Ax$. Then it has the form

$$(\tau_1(\overline{x}) = \tau_1'(\overline{x}) \ \& \ \cdots \ \& \ \tau_n(\overline{x}) = \tau_n'(\overline{x})) \Rightarrow \tau_0(\overline{x}) = \tau_0'(\overline{x}).$$

Let $\mathfrak{A} \overset{\text{def}}{=} (\underset{\sim}{F}/\sim_\Sigma)$. We want to prove that, for every valuation k of the variables into $\mathfrak{A}$, $\mathfrak{A} \models q[k]$.

So let k be an arbitrary valuation into $\mathfrak{A}$. Then $(\forall i \in \omega)\ k(x_i) = \varphi_i/{\sim_\Sigma}$ for some $\varphi_i \in F$. Assume that

$$\mathfrak{A} \models \tau_1\left(\overline{\varphi/{\sim_\Sigma}}\right) = \tau_1'\left(\overline{\varphi/{\sim_\Sigma}}\right) \ \& \ \cdots \ \& \ \tau_n\left(\overline{\varphi/{\sim_\Sigma}}\right) = \tau_n'\left(\overline{\varphi/{\sim_\Sigma}}\right).$$

Then

$$\mathfrak{A} \models (\tau_1(\overline{\varphi}))/\sim_\Sigma = (\tau_1'(\overline{\varphi}))/\sim_\Sigma \ \& \ \cdots \ \& \ (\tau_n(\overline{\varphi}))/\sim_\Sigma = (\tau_n'(\overline{\varphi}))/\sim_\Sigma,$$

since $\sim_\Sigma$ is a congruence on $\underset{\sim}{F}$. Then

$$\tau_1(\overline{\varphi}) \sim_\Sigma \tau_1'(\overline{\varphi}), \ldots, \tau_n(\overline{\varphi}) \sim_\Sigma \tau_n'(\overline{\varphi}),$$

that is,

$$\Sigma \vdash_{Ax} \tau_1(\overline{\varphi}) \nabla \tau_1'(\overline{\varphi}), \ldots, \Sigma \vdash_{Ax} \tau_n(\overline{\varphi}) \nabla \tau_n'(\overline{\varphi})$$

by the definition of $\sim_\Sigma$. In $\vdash_{Ax}$, we have the following rule (corresponding to q):

$$\frac{\tau_1(\overline{\Phi})\nabla\tau_1'(\overline{\Phi}), \ldots, \tau_n(\overline{\Phi})\nabla\tau_n'(\overline{\Phi})}{\tau_0(\overline{\Phi})\nabla\tau_0'(\overline{\Phi})}.$$

Using this rule, we obtain that $\Sigma \vdash_{Ax} \tau_0(\overline{\varphi}) \nabla \tau_0'(\overline{\varphi})$. Then $\tau_0(\overline{\varphi}) \sim_\Sigma \tau_0'(\overline{\varphi})$, whence $\mathfrak{A} \models (\tau_0(\overline{\varphi}))/\sim_\Sigma = (\tau_0'(\overline{\varphi}))/\sim_\Sigma$ that is, $\mathfrak{A} \models \tau_0\left(\overline{\varphi/\sim_\Sigma}\right) = \tau_0'\left(\overline{\varphi/\sim_\Sigma}\right)$ that is, $\mathfrak{A} \models \tau_0(\overline{x}) = \tau_0'(\overline{x})[k]$. By this we proved Claim 3.2.1.1. ▮

Claim 3.2.1.2: For any formulas $\varphi_0, \varphi_1, \ldots, \varphi_n$

$$\{\varphi_1 \ldots, \varphi_n\} \models \varphi_0 \implies \mathsf{Alg}_2(\mathcal{L}) \models (\varphi_1 = \mathit{True} \ \& \ \cdots \ \& \ \varphi_n = \mathit{True}) \Rightarrow (\varphi_0 = \mathit{True}).$$

PROOF OF CLAIM 3.2.1.2: Assume

$$\{\varphi_1(p_1, \ldots, p_m), \ldots, \varphi_n(p_1, \ldots, p_m)\} \models \varphi_0(p_1, \ldots, p_m).$$

Let $\mathfrak{A} \in \mathsf{Alg}_2(L)$. Then $\mathfrak{A} = mean_{\mathfrak{M}}(\underset{\sim}{F})$ for some $\mathfrak{M} \in M$. Let $k \in {}^P A$ be arbitrary. For every $1 \leq j \leq m$ we denote $k_j \overset{\text{def}}{=} k(p_j)$. Clearly for every $1 \leq j \leq m \quad k_j = mean_{\mathfrak{M}}(\gamma_j)$ for some $\gamma_j \in F$. For every $0 \leq i \leq n$

$$\varphi_i[k_1, \ldots, k_m]^{\mathfrak{A}} = \varphi_i[mean_{\mathfrak{M}}(\gamma_1), \ldots, mean_{\mathfrak{M}}(\gamma_m)]^{\mathfrak{A}} = mean_{\mathfrak{M}}(\varphi_i(\gamma_1, \ldots, \gamma_m)),$$

since $\mathit{mean}_{\mathfrak{M}}$ is a homomorphism.

Assume for every $1 \le i \le n$ that $\mathfrak{A} \models \varphi_i = \mathit{True}[k]$.

$$\begin{aligned}
&\iff \mathit{mean}_{\mathfrak{M}}(\varphi_i(\gamma_1, \ldots, \gamma_m)) = \mathit{mean}_{\mathfrak{M}}(\mathit{True}) \quad (1 \le i \le n) \\
\text{(by Def.3.1.1 (5))} &\implies (\exists \mathfrak{N})\ \mathit{mean}_{\mathfrak{N}}(\varphi_i) = \mathit{mean}_{\mathfrak{N}}(\mathit{True}) \quad (1 \le i \le n) \\
\text{(by Def.3.1.1 (3))} &\iff \mathfrak{N} \models \varphi_i \quad (1 \le i \le n) \\
\text{(by our assumption)} &\implies \mathfrak{N} \models \varphi_0 \\
\text{(by Def.3.1.1 (3))} &\iff \mathit{mean}_{\mathfrak{N}}(\varphi_0) = \mathit{mean}_{\mathfrak{N}}(\mathit{True}) \\
\text{(by Def.3.1.1 (5))} &\implies \mathit{mean}_{\mathfrak{M}}(\varphi_0(\gamma_1, \ldots, \gamma_m)) = \mathit{mean}_{\mathfrak{M}}(\mathit{True}) \\
&\iff \mathfrak{A} \models \varphi_0 = \mathit{True}[k],
\end{aligned}$$

proving Claim 3.2.1.2, since k was chosen arbitrarily. ∎

Now let $\Sigma \overset{\text{def}}{=} \{\varphi_1, \ldots, \varphi_n\}$ and assume $\Sigma \models \varphi_0$. Then, by Claim 3.2.1.2,

$$\begin{aligned}
&\mathsf{Alg}_2(\mathcal{L}) \models (\varphi_1 = \mathit{True}\ \&\ \cdots\ \&\ \varphi_n = \mathit{True}) \Rightarrow (\varphi_0 = \mathit{True}) \\
&\implies \mathit{Ax} \models (\varphi_1 = \mathit{True}\ \&\ \cdots\ \&\ \varphi_n = \mathit{True}) \Rightarrow (\varphi_0 = \mathit{True}) \\
&\overset{\text{(Claim 3.2.1.1)}}{\implies} \left(\underset{\sim}{F}/{\sim_\Sigma}\right) \models (\varphi_1 = \mathit{True}\ \&\ \cdots\ \&\ \varphi_n = \mathit{True}) \Rightarrow (\varphi_0 = \mathit{True}) \\
&\implies \left[\text{if } (\varphi_1 \sim_\Sigma \mathit{True}, \ldots, \varphi_n \sim_\Sigma \mathit{True}) \text{ then } \varphi_0 \sim_\Sigma \mathit{True}\right] \\
&\iff \left[\text{if } (\Sigma \vdash_{Ax} \varphi_1 \nabla \mathit{True}, \ldots, \Sigma \vdash_{Ax} \varphi_n \nabla \mathit{True}) \text{ then } \Sigma \vdash_{Ax} \varphi_0 \nabla \mathit{True}\right] \qquad (\bullet).
\end{aligned}$$

By the rule $\frac{\Phi_0}{\Phi_0 \nabla \mathit{True}}$ we have $\Sigma \vdash_{Ax} \varphi_1 \nabla \mathit{True}, \ldots, \Sigma \vdash_{Ax} \varphi_n \nabla \mathit{True}$. Thus, by $(\bullet)$, $\Sigma \vdash_{Ax} \varphi_0 \nabla \mathit{True}$. Now using the rule $\frac{\Phi_0 \nabla \mathit{True}}{\Phi_0}$ we get $\Sigma \vdash_{Ax} \varphi_0$, proving the finite completeness of $\vdash_{Ax}$.

The strong soundness of $\vdash_{Ax}$ can be proved by induction on the length of the $\vdash_{Ax}$-proof of φ_0 from $\{\varphi_1, \ldots, \varphi_n\}$. We only show one part of the induction step, namely the case when φ_0 is 'obtained' by an inference rule corresponding to a quasiequation $q \in Ax$. Say q has the form

$$(\tau_1(\bar{x}) = \tau_1'(\bar{x})\ \&\ \cdots\ \&\ \tau_k(\bar{x}) = \tau_k'(\bar{x})) \Rightarrow \tau_0(\bar{x}) = \tau_0'(\bar{x}),$$

where $\bar{x} = \langle x_1, \ldots, x_m \rangle$. Then the corresponding inference rule is

$$\frac{\tau_1(\bar{\Phi}) \nabla \tau_1'(\bar{\Phi}), \ldots, \tau_k(\bar{\Phi}) \nabla \tau_k'(\bar{\Phi})}{\tau_0(\bar{\Phi}) \nabla \tau_0'(\bar{\Phi})}.$$

Assume that φ_0 is obtained by this rule by substituting the members of the sequence $\bar{\gamma} = \langle \gamma_1, \ldots, \gamma_m \rangle$ of formulas for the members of the sequence $\bar{\Phi} = \langle \Phi_1, \ldots, \Phi_m \rangle$ of formula variables, i.e. φ_0 has the form $\tau_0(\bar{\gamma}) = \tau_0'(\bar{\gamma})$.

Now fix a model $\mathfrak{M}$ and assume that

$$\mathfrak{M} \models \tau_1(\bar{\gamma}) \nabla \tau_1'(\bar{\gamma}), \ldots, \mathfrak{M} \models \tau_k(\bar{\gamma}) \nabla \tau_k'(\bar{\gamma}).$$

We have to show that $\mathfrak{M} \models \tau_0(\overline{\gamma})\nabla\tau_0'(\overline{\gamma})$.

Let $\mathfrak{A} \overset{\text{def}}{=} \mathit{mean}_{\mathfrak{M}}(\underset{\sim}{F}) \in \mathsf{Alg}_2(\mathcal{L})$. and let k be a valuation of $\mathfrak{A}$ such that for every $1 \leq i \leq m \quad k(x_i) \overset{\text{def}}{=} \mathit{mean}_{\mathfrak{M}}(\gamma_i)$. Then by Definition 3.1.1 (3)(i)

$$\begin{aligned} & \mathit{mean}_{\mathfrak{M}}(\tau_j(\overline{\gamma})) = \mathit{mean}_{\mathfrak{M}}(\tau_j'(\overline{\gamma})) \quad (1 \leq j \leq k) \\ & \iff \mathfrak{A} \models \tau_1(\overline{x}) = \tau_1'(\overline{x}) \ \& \ \cdots \ \& \ \tau_k(\overline{x}) = \tau_k'(\overline{x})[k] \\ \text{(by } \mathsf{Alg}_2(\mathcal{L}) \models Ax\text{)} \quad & \Longrightarrow \mathfrak{A} \models \tau_0(\overline{x}) = \tau_0'(\overline{x})[k] \\ \text{(by Def.3.1.1 (3)(i))} \quad & \iff \mathfrak{M} \models \tau_0(\overline{\gamma})\nabla\tau_0'(\overline{\gamma}). \end{aligned}$$

This completes the proof of direction "$\Longrightarrow$" of Theorem 3.2.1.

Proof of ($\Longleftarrow$): Let $\Phi_1, \ldots, \Phi_m$ denote formula variables, $\tau_0, \tau_1, \ldots, \tau_k$ denote formula schemes, let $\overline{\Phi} \overset{\text{def}}{=} \langle \Phi_1, \ldots, \Phi_m \rangle$, and let $\overline{x} \overset{\text{def}}{=} \langle x_1, \ldots, x_m \rangle$ be a sequence of variables. Assume that $\vdash$ is a finitely complete and strongly sound Hilbert–type inference system for the logic $\mathcal{L}$, and define the finite set Ax of quasiequations as follows:

- If $\tau_0(\overline{\Phi})$ is an axiom scheme of $\vdash$ then let "$\tau_0(\overline{x}) = \mathit{True}$" belong to Ax.
- If $\frac{\tau_1(\overline{\Phi}), \ldots, \tau_k(\overline{\Phi})}{\tau_0(\Phi)}$ is an inference rule of $\vdash$ then let "$(\tau_1(\overline{x}) = \mathit{True} \ \& \ \cdots \ \& \ \tau_k(\overline{x}) = \mathit{True}) \Rightarrow \tau_0(\overline{x}) = \mathit{True}$" belong to Ax.
- Let "$(x_0 = x_1) \Rightarrow (x_0 \nabla x_1 = \mathit{True})$" and "$(x_0 \nabla x_1 = \mathit{True}) \Rightarrow (x_0 = x_1)$" belong to Ax.

We will show that Ax axiomatizes the quasivariety generated by $\mathsf{Alg}_2(\mathcal{L})$.

Claim 3.2.1.3: $\mathsf{Alg}_2(\mathcal{L}) \models Ax$.

PROOF OF CLAIM 3.2.1.3: $\mathsf{Alg}_2(\mathcal{L}) \models (x_0 \nabla x_1 = \mathit{True}) \Leftrightarrow (x_0 = x_1)$ obviously holds by Definition 3.1.1 (3).

Let $(\tau_1(\overline{x}) = \mathit{True} \ \& \ \cdots \ \& \ \tau_k(\overline{x}) = \mathit{True}) \Rightarrow \tau_0(\overline{x}) = \mathit{True}$ belong to Ax, let $\mathfrak{A} \in \mathsf{Alg}_2(\mathcal{L})$ and let k be an arbitrary valuation of the variables into $\mathfrak{A}$. Let $\mathfrak{M}$ be such that $\mathfrak{A} = \mathit{mean}_{\mathfrak{M}}(\underset{\sim}{F})$. Then for every $i \in \omega \quad k(x_i) = \mathit{mean}_{\mathfrak{M}}(\varphi_i)$ for some $\varphi_i \in F$. Assume that

$$\mathfrak{A} \models \tau_1(\overline{x}) = \mathit{True} \ \& \ \cdots \ \& \ \tau_k(\overline{x}) = \mathit{True}[k].$$

Then by Definition 3.1.1 (3)

$$(\bullet\bullet) \qquad \mathfrak{M} \models \tau_j(x_1/\varphi_1, \ldots, x_m/\varphi_m) \quad (1 \leq j \leq k).$$

But $\frac{\tau_1(\overline{\Phi}), \ldots, \tau_k(\overline{\Phi})}{\tau_0(\Phi)}$ is an inference rule of $\vdash$, therefore $\{\tau_1(\overline{\varphi}), \ldots, \tau_k(\overline{\varphi})\} \vdash \tau_0(\overline{\varphi})$. This implies by the strong soundness of $\vdash$ that $\{\tau_1(\overline{\varphi}), \ldots, \tau_n(\overline{\varphi})\} \models \tau_0(\overline{\varphi})$. Now, by $(\bullet\bullet)$ above, $\mathfrak{M} \models \tau_0(\overline{\varphi})$, hence again by Definition 3.1.1 (3), $\mathfrak{A} \models \tau_0(\overline{x}) = \mathit{True}[k]$, which is desired. ∎

Claim 3.2.1.4: For any quasiequation q of form $\tau_1 = \tau_1' \ \& \ \cdots \ \& \ \tau_n = \tau_n' \Rightarrow \tau_0 = \tau_0'$

$$\mathsf{Alg}_2(\mathcal{L}) \models q \implies \{\tau_1 \nabla \tau_1', \ldots, \tau_n \nabla \tau_n'\} \models \tau_0 \nabla \tau_0'.$$

Proof of Claim 3.2.1.4: Assume that for every $\mathfrak{A} \in \mathsf{Alg}_2(L)$ and for every valuation $k \in {}^P A$

$$\mathfrak{A} \models q[k].$$

Let $\mathfrak{M} \in M$ such that $\mathfrak{M} \models \{\tau_1 \nabla \tau_1', \ldots, \tau_n \nabla \tau_n'\}$. Then by Definition 3.1.1 (3)(i) $mean_{\mathfrak{M}}(\tau_i) = mean_{\mathfrak{M}}(\tau_i')$ for each $1 \leq i \leq n$. Now let $\mathfrak{A} \in \mathsf{Alg}_2(L)$ be such that $mean_{\mathfrak{M}}(\underset{\sim}{F}) = \mathfrak{A}$ and let $k \in {}^P A$ be such that for each $p \in P \quad k(p) \overset{\text{def}}{=} mean_{\mathfrak{M}}(p)$. Then

$$\mathfrak{A} \models (\tau_1 = \tau_1' \ \& \ \cdots \ \& \ \tau_n = \tau_n')[k],$$

which implies $\mathfrak{A} \models (\tau_0 = \tau_0')[k]$ by our assumption. This is the same as $mean_{\mathfrak{M}}(\tau_0) = mean_{\mathfrak{M}}(\tau_0')$, thus again by Definition 3.1.1 (3)(i), $\mathfrak{M} \models \tau_0 \nabla \tau_0'$, which proves Claim 3.2.1.4. ▮

Claim 3.2.1.5: For any formulas $\varphi_0, \varphi_1, \ldots, \varphi_n$

$$\{\varphi_1 \ldots, \varphi_n\} \vdash \varphi_0 \implies Ax \models (\varphi_1 = True \ \& \ \cdots \ \& \ \varphi_n = True) \Rightarrow (\varphi_0 = True).$$

Proof of Claim 3.2.1.5: It can be proved by induction on the length of the $\vdash$–proof of φ_0 from $\{\varphi_1, \ldots, \varphi_n\}$. We only show one part of the induction step, namely the case when φ_0 is 'obtained' by an inference rule $\frac{\tau_1(\overline{\Phi}), \ldots, \tau_k(\overline{\Phi})}{\tau_0(\overline{\Phi})}$, where $\overline{\Phi} = \langle \Phi_1, \ldots, \Phi_m \rangle$. Then there are formulas $\gamma_1, \ldots, \gamma_m$ such that $\varphi_0 = \tau_0(\gamma_1, \ldots, \gamma_m)$ and for every $1 \leq i \leq k$ $\{\varphi_1, \ldots, \varphi_n\} \vdash \tau_i(\overline{\gamma})$. Then by the induction hypothesis

$$(\natural) \qquad Ax \models (\varphi_1 = True \ \& \ \cdots \ \& \ \varphi_n = True) \Rightarrow \tau_i(\overline{\gamma}) \quad (1 \leq i \leq k).$$

By the definition of Ax

$$(\natural\natural) \qquad Ax \models (\tau_1(\overline{x}) = True \ \& \ \cdots \ \& \ \tau_k(\overline{x}) = True) \Rightarrow \tau_0(\overline{x}) = True.$$

Let $\mathfrak{B}$ be an algebra with $\mathfrak{B} \models Ax$ and let k be any valuation of the variables into B. Now we can define a valuation k' with $k'(x_j) \overset{\text{def}}{=} \gamma_j[k]^{\mathfrak{B}} \quad (1 \leq j \leq m)$. Then for every $0 \leq i \leq k \quad \tau_i(\overline{x})[k']^{\mathfrak{B}} = \tau_i(\overline{\gamma})[k]^{\mathfrak{B}}$. Thus, by $(\natural)$ and $(\natural\natural)$,

$$\mathfrak{B} \models (\varphi_1 = True \ \& \ \cdots \ \& \ \varphi_n = True) \Rightarrow \tau_0(\overline{\gamma}) = True[k],$$

which was desired. ▮

Now assume that

$$\mathsf{Alg}_2(\mathcal{L}) \models (\tau_1 = \tau_1' \ \& \ \cdots \ \& \ \tau_n = \tau_n') \Rightarrow \tau_0 = \tau_0'.$$

$$\overset{\text{(Claim 3.2.1.4)}}{\Longrightarrow} \{\tau_1 \nabla \tau_1', \ldots, \tau_n \nabla \tau_n'\} \models \tau_0 \nabla \tau_0'$$

$$\overset{\text{(finite completeness)}}{\Longrightarrow} \{\tau_1 \nabla \tau_1', \ldots, \tau_n \nabla \tau_n'\} \vdash \tau_0 \nabla \tau_0'$$

$$\overset{\text{(Claim 3.2.1.5)}}{\Longrightarrow} Ax \models (\tau_1 \nabla \tau_1' = True \ \& \ \cdots \ \& \ \tau_n \nabla \tau_n' = True) \Rightarrow \tau_0 \nabla \tau_0' = True.$$

But, since "$x_0 \nabla x_1 = \mathit{True} \Leftrightarrow x_0 = x_1$" belongs to Ax, this is equivalent to

$$Ax \models (\tau_1 = \tau_1' \ \& \ \cdots \ \& \ \tau_n = \tau_n') \Rightarrow \tau_0 = \tau_0',$$

completing the proof of direction "$\Longleftarrow$" of Theorem 3.2.1. ∎

Having found the algebraic counterpart of "finitely complete", let us try to characterize "weakly complete". Since weak completeness is slightly weaker than finite completeness, we have to weaken the algebraic counterpart of finite completeness for characterizing weak completeness. This way we obtain condition $(*)$ below, where $Eq_{\mathcal{L}}$ and $Qeq_{\mathcal{L}}$ denote the set of all equations and the set of all quasiequations, respectively, of the language of $\mathsf{Alg}_2(\mathcal{L})$.

$$(*) \qquad (\exists Ax \subseteq_\omega Qeq_{\mathcal{L}})\left[(\forall e \in Eq_{\mathcal{L}})(\mathsf{Alg}_2(\mathcal{L}) \models e \Longrightarrow Ax \models e) \ \& \ \mathsf{Alg}_2(\mathcal{L}) \models Ax\right].$$

THEOREM 3.2.2. *Assume that $\mathcal{L}$ is nice. Then*

$$(*) \Longleftrightarrow (\exists \textit{ Hilbert–style} \vdash)(\vdash \textit{ is weakly complete and strongly sound for } \mathcal{L}).$$

In particular, if the equational theory of $\mathsf{Alg}_2(\mathcal{L})$ is finitely axiomatizable, then $\mathcal{L}$ admits a weakly complete Hilbert–style inference system.

Proof: It is similar to the proof of Theorem 3.2.1. The only important difference is that Theorem 3.2.2 already holds for *nice* logics. However, the only part of the proof of Theorem 3.2.1 which used the additional criterion for strong niceness (Definition 3.1.1 (5)) was Claim 3.2.1.2. Below we state the corresponding weaker claim and prove it without using condition (5) of Definition 3.1.1.

Claim 3.2.2.2: For any formula φ

$$\models \varphi \Longrightarrow \mathsf{Alg}_2(\mathcal{L}) \models (\varphi = \mathit{True}).$$

PROOF OF CLAIM 3.2.2.2: Assume $\models \varphi(p_0, \ldots, p_n)$. Let $\mathfrak{A} \in \mathsf{Alg}_2(L)$. Then $\mathfrak{A} = \mathit{mean}_{\mathfrak{M}}(\underset{\sim}{F})$ for some $\mathfrak{M} \in M$. Let $k \in {}^{P}A$ be arbitrary. We denote $k_0 \overset{\text{def}}{=} k(p_0), \ldots, k_n \overset{\text{def}}{=} k(p_n)$. Clearly $(\forall i \le n)(k_i = \mathit{mean}_{\mathfrak{M}}(\gamma_i)$ for some $\gamma_i \in F)$.

$$\varphi[k_0, \ldots, k_n]^{\mathfrak{A}} = \varphi[\mathit{mean}_{\mathfrak{M}}(\gamma_0), \ldots, \mathit{mean}_{\mathfrak{M}}(\gamma_n)]^{\mathfrak{A}} = \mathit{mean}_{\mathfrak{M}}(\varphi(\gamma_0, \ldots, \gamma_n)),$$

since $\mathit{mean}_{\mathfrak{M}}$ is a homomorphism.

$\models \varphi(p_0, \ldots, p_n)$ implies, by Definition 3.1.1.(4) (substitution property), that $\models \varphi(\gamma_0, \ldots, \gamma_n)$. Thus by Definition 3.1.1 (3)

$$\mathit{mean}_{\mathfrak{M}}(\varphi(\gamma_0, \ldots, \gamma_n)) = \mathit{mean}_{\mathfrak{M}}(\mathit{True}).$$

But $\mathit{mean}_{\mathfrak{M}}(\varphi(\gamma_0, \ldots, \gamma_n)) = \varphi[k_0, \ldots, k_n]^{\mathfrak{A}}$ and $\mathit{mean}_{\mathfrak{M}}(\mathit{True}) = \mathit{True}^{\mathfrak{A}}$, thus $\varphi[k_0, \ldots, k_n]^{\mathfrak{A}} = \mathit{True}^{\mathfrak{A}}$. Thus we have $\mathfrak{A} \models (\varphi = \mathit{True})[k]$, proving Claim 3.2.2.2, since k was chosen arbitrarily. Thus we also proved Theorem 3.2.2. ∎

EXERCISE 3.2.1: Give weakly complete and sound calculi for the logics $\mathcal{L}_S$, $S5$, $\mathcal{L}_{\mathrm{ARW0}}$ and $\mathcal{L}_{\mathrm{ARWRL}}$. (Hint: Use that the **SP**–closure of the Alg_2–image of these logics are finitely axiomatizable varieties, so $(*)$ is satisfied. For the arrow logics, finite axiomatizability of the corresponding varieties will be prcved in chapter I) ◀

DEFINITION 3.2.1: Let $\mathcal{L} = \langle F, M, \models\rangle$ be a nice logic. We say that $\mathcal{L}$ *has a deduction theorem*, iff

$$(\exists(\Phi_1\Delta\Phi_2) \in Fms_{\mathcal{L}}))(\forall\Sigma \subseteq F)(\forall\varphi,\psi \in F)(\Sigma \cup \{\varphi\} \models \psi \iff \Sigma \models \varphi\Delta\psi)\,,$$

where "$\varphi\Delta\psi$" denotes an instance of scheme "$\Phi_1\Delta\Phi_2$". Such a "$\Phi_1\Delta\Phi_2$" is called a *deduction term* for $\mathcal{L}$. ◀

THEOREM 3.2.3. *$\mathcal{L}_S$ and $S5$ have deduction terms.*

Proof: It is not hard to show that "$\Phi_1 \to \Phi_2$" and "$\Box\Phi_1 \to \Box\Phi_2$" (where $\Box$ is the abbreviation of $\neg\Diamond\neg$) are suitable deduction terms for propositional logic and $S5$, respectively. ▮

The following theorem states that for any nice logic the existence of a deduction term and that of a weakly complete Hilbert–style calculus provides a finitely complete inference system.

THEOREM 3.2.4. *Assume $\mathcal{L}$ has a deduction theorem, and ($\exists$ Hilbert–style $\vdash$) ($\vdash$ is weakly complete and strongly sound for $\mathcal{L}$). Then*

$$(\exists \text{ Hilbert-style } \vdash)(\vdash \text{ is finitely complete and strongly sound for } \mathcal{L})\,.$$

First we note the following fact (its proof is straightforward by the assumptions on Δ).

Fact 3.2.4.1: The inference rule *modus ponens w.r.t.* Δ (MP_Δ) that is,

$$(\mathrm{MP}_\Delta) \qquad \frac{\Phi,\ \Phi\Delta\Psi}{\Psi}$$

is strongly sound for $\mathcal{L}$. ▮

Proof of Theorem 3.2.4:

Assume that ($\exists$ Hilbert–style $\vdash$)($\vdash$ is weakly complete and strongly sound for $\mathcal{L}$). Let such an inference system be fixed and let us add (MP_Δ) to it. We denote this (extended) inference system by $\vdash$, too.

To prove finite completeness, assume $\{\varphi_0, \ldots, \varphi_n\} \models \psi$. Then, applying the deduction theorem $n+1$ times, we get:

$$\begin{aligned}
\{\varphi_0, \ldots, \varphi_{n-1}\} &\models (\varphi_n\Delta\psi)\\
\{\varphi_0, \ldots, \varphi_{n-2}\} &\models (\varphi_{n-1}\Delta(\varphi_n\Delta\psi))\\
&\vdots\\
&\models \underbrace{(\varphi_0\Delta(\varphi_1\Delta\ldots(\varphi_n\Delta\psi)\ldots)}_{\gamma_0}.
\end{aligned}$$

Then $\vdash \gamma_0$ by weak completeness of $\vdash$. Then, using (MP_Δ) $n+1$ times, we get:

$$\begin{aligned}
\{\varphi_0\} &\vdash \{\varphi_0,\gamma_0\} \vdash \underbrace{\varphi_1\Delta(\varphi_2\Delta\ldots(\varphi_n\Delta\psi)\ldots)}_{\gamma_1}\\
\{\varphi_0,\varphi_1\} &\vdash \{\varphi_1,\gamma_1\} \vdash \underbrace{\varphi_2\Delta(\varphi_2\Delta\ldots(\varphi_n\Delta\psi)\ldots)}_{\gamma_2}\\
&\vdots \qquad\qquad \vdots\\
\{\varphi_0,\varphi_1,\ldots,\varphi_n\} &\vdash \{\varphi_n,\gamma_n\} \vdash \psi \quad, \text{ where } \gamma_n = (\varphi_n\Delta\psi).
\end{aligned}$$

Thus we received the following $\vdash$–proof of ψ from $\{\varphi_0,\ldots,\varphi_n\}$:

$$\langle\gamma_0,\varphi_0,\gamma_1,\varphi_1,\gamma_2,\varphi_2,\ldots,\gamma_n\varphi_n,\psi\rangle,$$

which proves Theorem 3.2.4. ▮

DEFINITION 3.2.2: Let $\mathcal{L} = \langle F, M, \models\rangle$ be a logic. We say that

(i) $\mathcal{L}$ is *satisfiability compact* (sat. compact for short), if

$$(\forall\Gamma \subseteq F)\big[(\forall\Sigma \subseteq_\omega \Gamma)(\Sigma \text{ has a model}) \Longrightarrow (\Gamma \text{ has a model})\big]\text{, and}$$

(ii) $\mathcal{L}$ is *consequence compact* (cons. compact), if

$$\Gamma \models \varphi \implies (\exists\Sigma \subseteq_\omega \Gamma)\ \Sigma \models \varphi, \quad \text{for every } \Gamma\cup\{\varphi\} \subseteq F. ◀$$

◀

EXERCISE 3.2.2: Prove that even for nice logics we have

(1) satisfiability compact $\not\Longrightarrow$ consequence compact;
(2) satisfiability compact $\not\Longleftarrow$ consequence compact.

(Hint for (1): Let the logical connectives be ∇ (binary), and $\mathit{True}, k_0,\ldots,k_n,\ldots$ all zero–ary. A model $\mathfrak{M}$ is a function $\mathfrak{M} : \{\mathit{True}, p_i, k_i : i \in \omega\} \to \{0,1\}$. $\mathit{mean}_{\mathfrak{M}}(\mathit{True}) = 1$ for every $\mathfrak{M}$ and meaning of ∇ is the standard meaning of the biconditional $\leftrightarrow$. Exclude those models from M in which $(\forall\ i > 0)\ \mathfrak{M}(k_i) = 1$ but $\mathfrak{M}(k_0) = 0$. [This logic is not strongly nice!] Observe that for $\mathfrak{M} = \{\mathit{True}, p_i, k_i : i \in \omega\} \times \{1\}$ we have $\mathfrak{M} \models F_{\mathcal{L}}$. Hence sat. completeness trivially holds.)

(Hint for (2): Let $\mathcal{L}$ have *True* and ∇ as the only logical connectives. Exclude the models $\mathfrak{M}$ with $\mathfrak{M} \models F_{\mathcal{L}}$. Then sat. completeness fails (we have infinitely many propositional variables). Show that cons. completeness remains true.) ◀

EXERCISE 3.2.3: Find natural conditions under which "$\Longrightarrow$" and/or "$\Longleftarrow$" of Exercise 3.2.2 above hold.

(1) We say that $\mathcal{L}$ *has weak false* if $(\exists\varphi \in F_{\mathcal{L}})$ such that $(\forall\mathfrak{M} \in M)\ \mathfrak{M} \not\models \varphi$. Show that under this assumption

$$\text{cons. compact} \implies \text{sat. compact.}$$

(2) We say that $\mathcal{L}$ *has negation* if

$$(\forall\varphi \in F)(\exists\psi \in F)(\forall\mathfrak{M} \in M)[\mathfrak{M} \models \psi \iff \mathfrak{M} \not\models \varphi].$$

Show that under this assumption

$$\text{sat. compact} \implies \text{cons. compact.}$$

(3) Try to find weaker sufficient conditions.

(4) Show that for nice logics

$$\mathcal{L} \text{ has weak false} \iff \mathcal{L} \text{ has negation.}$$

For the whole matter [ANS84] might contain useful info. ◄

* * *

Recall that in Definition 3.1.1 above (and also in the logics studied so far), there was a parameter P, which was the set of atomic formulas. The choice of P influenced what the set F of formulas would be. Thus in fact, our old definition of a logic yields a family

$$\{\langle F^P, M^P, \models^P\rangle : P \text{ is a set}\}$$

of logics. The members of this family do not differ significantly except that the cardinality of P matters sometimes.

DEFINITION 3.2.3: (General Logic)

A *general logic* is a class

$$\mathbf{L} \stackrel{\text{def}}{=} \{\mathcal{L}^\alpha : \alpha \text{ is a cardinal}\},$$

where for each cardinal α $\mathcal{L}^\alpha = \langle F^\alpha, M^\alpha, \models^\alpha\rangle$ is a logic in the sense of Definition 2.1 that is, F^α is a set, M^α is a class, and $\models^\alpha \subseteq M^\alpha \times F^\alpha$.

$\mathbf{L}$ is called a *(strongly) nice general logic* iff conditions (1–3) below hold for $\mathbf{L}$.

(1) $\mathcal{L}^\alpha$ is a (strongly) nice logic (cf. Def.3.1.1) for each cardinal α.

(2) For each cardinal α the set P^α of atomic formulas of the logic $\mathcal{L}^\alpha$ is of cardinality α. If α and λ are cardinals with $\lambda \leq \alpha$ then $P^\lambda \subseteq P^\alpha$ (which implies that $F^\lambda \subseteq F^\alpha$).

(3) For all cardinals $\lambda \leq \alpha$

$$\{mean^\lambda_{\mathfrak{M}} : \mathfrak{M} \in M^\lambda\} = \{(mean^\alpha_{\mathfrak{M}}) \restriction F^\lambda : \mathfrak{M} \in M^\alpha\},$$

(cf. item (2) of Def.3.1.1 for *mean*). Intuitively, this requirement says that $\mathcal{L}^\lambda$ is the "natural" restriction of $\mathcal{L}^\alpha$. ◄

REMARK 3.2.3.1: As a corollary of item (3) of Definiton 3.2.3 above we note that for all cardinal α, λ, if $\Gamma \cup \{\varphi\} \subseteq F^\alpha \cap F^\lambda$ then

$$\Gamma \models^\alpha \varphi \iff \Gamma \models^\lambda \varphi. \qquad \blacktriangleleft$$

DEFINITION 3.2.4 (Algebraic Counterpart of a General Logic): Let $\mathbf{L} = \{\mathcal{L}^\alpha : \alpha \text{ is a cardinal}\}$ be a nice general logic. Then

$$\mathsf{Alg}_1(\mathbf{L}) \overset{\text{def}}{=} \bigcup \{\mathsf{Alg}_1(\mathcal{L}^\alpha) \ : \ \alpha \text{ is a cardinal}\},$$
$$\mathsf{Alg}_2(\mathbf{L}) \overset{\text{def}}{=} \bigcup \{\mathsf{Alg}_2(\mathcal{L}^\alpha) \ : \ \alpha \text{ is a cardinal}\}$$

(cf. Def.3.1.2). $\blacktriangleleft$

THEOREM 3.2.5. *For strongly nice general logics*

$$\mathsf{Alg}_1(\mathbf{L}) = \mathbf{SP}\mathsf{Alg}_2(\mathbf{L}).$$

Proof: First we prove that for any nice logic $\mathcal{L} = \langle F, M, \models\rangle$, $\mathsf{Alg}_1(\mathcal{L}) \subseteq \mathsf{Alg}_2(\mathcal{L})$. We note that if K is a subclass of M then there is a subset $K' \subseteq K$ such that $\underset{\sim}{F}/\sim_K = \underset{\sim}{F}/\sim_{K'}$ (this holds because F is always a set). Now let K be any subclass of M and let $K' \subseteq K$ has the property above. Then function h below is a one–one homomorphism (i.e. an embedding) of $\underset{\sim}{F}/\sim_K$ into $\mathbf{P}_{\mathfrak{M}\in K'} mean_{\mathfrak{M}}(\underset{\sim}{F})$. For each $\varphi \in F$

$$h(\varphi/\sim_K) \overset{\text{def}}{=} \langle mean_{\mathfrak{M}}(\varphi) \ : \ \mathfrak{M} \in K'\rangle.$$

Next we prove that $\mathbf{SP}\mathsf{Alg}_2(\mathbf{L}) \subseteq \mathsf{Alg}_1(\mathbf{L})$. Assume $\mathfrak{A} \subseteq \mathbf{P}_{\mathfrak{M}\in K} mean^\lambda_{\mathfrak{M}}(\underset{\sim}{F}^\lambda)$ for some cardinal λ and set $K \subseteq M^\lambda$. Let $\alpha \overset{\text{def}}{=} |\mathfrak{A}|$, fix any bijection from the set P^α of atomic formulas of $\mathcal{L}^\alpha$ onto A and let $h : \underset{\sim}{F}^\alpha \twoheadrightarrow \mathfrak{A}$ be its natural extension to a homomorphism *onto* $\mathfrak{A}$.

Claim 3.2.5.1: For every $\mathfrak{M} \in K$ there is some $\mathfrak{N} \in M^\alpha$ such that

$$(\forall p \in P^\alpha) \ mean^\alpha_{\mathfrak{N}}(p) = h(p)_{\mathfrak{M}},$$

where $h(p)_{\mathfrak{M}}$ denotes the $\mathfrak{M}^{\text{th}}$ member of the sequence $h(p)$.

PROOF OF CLAIM 3.2.5.1: Fix any $\mathfrak{M} \in K$ and assume that $h(p)_{\mathfrak{M}} = mean^\lambda_{\mathfrak{M}}(\gamma^p_{\mathfrak{M}})$ for some formula $\gamma^p_{\mathfrak{M}} \in F^\lambda$. Then, by (3) of Def.3.2.3, there is some $\mathfrak{N}' \in M^\alpha$ with $mean^\lambda_{\mathfrak{M}}(\gamma^p_{\mathfrak{M}}) = mean^\alpha_{\mathfrak{N}'}(\gamma^p_{\mathfrak{M}})$. Let $s_{\mathfrak{M}} : P^\alpha \to F^\alpha$ be the substitution defined by

$$(\bullet) \qquad s_{\mathfrak{M}}(p) \overset{\text{def}}{=} \gamma^p_{\mathfrak{M}}.$$

Then condition (5) of Def.3.1.1 gives a model $\mathfrak{N} \in M^\alpha$ with $mean^\alpha_{\mathfrak{N}}(p) = mean^\alpha_{\mathfrak{N}'}(\gamma^p_{\mathfrak{M}})$. ∎

Now for each $\mathfrak{M} \in K$ we can define a nonempty class $M(\mathfrak{M}) \subseteq M^\alpha$ as follows.

$$M(\mathfrak{M}) \stackrel{\text{def}}{=} \{\mathfrak{N} \in M^\alpha \ : \ (\forall p \in P^\alpha)\ mean^\alpha_{\mathfrak{N}}(p) = h(p)_{\mathfrak{M}}\}.$$

Let $K' \stackrel{\text{def}}{=} \bigcup\{M(\mathfrak{M}) : \mathfrak{M} \in K\}$.
Claim 3.2.5.2: $\underset{\sim}{F}^\alpha / \sim_{K'} \cong \mathfrak{A}$.

PROOF OF CLAIM 3.2.5.2: Fix an $\mathfrak{M} \in K$ and let $s_{\mathfrak{M}}$ be the substitution in $(\bullet)$ above. It can be proved by induction on the complexity of formulas that for any formula $\varphi \in F^\alpha$ and for any $\mathfrak{N} \in M(\mathfrak{M})$

$$h(\varphi)_{\mathfrak{M}} = mean^\lambda_{\mathfrak{M}}(\varphi(s_{\mathfrak{M}})) = mean^\alpha_{\mathfrak{N}}(\varphi),$$

where $\varphi(s_{\mathfrak{M}})$ is obtained from φ by substituting $s_{\mathfrak{M}}(p)$ for each atomic formula p occurring in φ. Now h gives the required isomorphism between $\underset{\sim}{F}^\alpha / \sim_{K'}$ and $\mathfrak{A}$, since for all formulas $\varphi, \psi \in F^\alpha$

$$h(\varphi) = h(\psi) \quad \text{iff} \quad \varphi \sim_{K'} \psi,$$

which proves Claim 3.2.5.2. ▮

Now, since $\underset{\sim}{F}^\alpha / \sim_{K'} \in \mathsf{Alg}_1(\mathcal{L}^\alpha)$, the proof of Theorem 3.2.5 is completed. ▮

DEFINITION 3.2.5: A general logic $\mathbf{L} = \{\mathcal{L}^\alpha : \alpha \text{ is a cardinal}\}$ is ***satisfiability (consequence) compact*** if for each cardinal α the logic $\mathcal{L}^\alpha$ is satisfiability (consequence) compact. ◀

For an arbitrary class K of algebras,

$$\mathsf{Up}\mathsf{K} \stackrel{\text{def}}{=} \mathbf{I}\,\{\mathbf{P}_{i \in I}\mathfrak{A}_i/\mathcal{F} : \mathcal{F} \text{ is an ultrafilter over the set } I, \text{ and } (\forall i \in I)\mathfrak{A}_i \in \mathsf{K}\}\ .$$

We say that K is **Up**–*closed* if $\mathsf{Up}\mathsf{K} \subseteq \mathsf{K}$, in other words, K is **Up**–closed if it is closed under taking ultraproducts.

Our next theorem gives a sufficent condition for sat. compactness of a general logic.

THEOREM 3.2.6. *Assume* $\mathbf{L}$ *is a strongly nice general logic. Then*

$$(\mathsf{Alg}_1(\mathbf{L}) \textit{ is } \mathbf{Up}\textit{–closed}) \Longrightarrow (\mathbf{L} \textit{ is sat. compact}).$$

Proof: We let $\mathbf{L} = \{\mathcal{L}^\alpha : \alpha \text{ is a cardinal}\}$ We give a proof for the compactness of $\mathcal{L}^\omega = \langle F^\omega, M^\omega, \models^\omega \rangle$. For other cardinals the proof is similar and is left to the reader. Assume $\Gamma \subseteq F^\omega$ and

$$(\forall \Sigma \subseteq_\omega \Gamma)\ \Sigma \text{ has a model}.$$

Then we may assume that $\Gamma = \{\varphi_0, \varphi_1, \ldots, \varphi_n, \ldots\}_{n \in \omega}$ and

$$(\forall k \in \omega)(\exists \mathfrak{M}_k \in M^\omega)\ \mathfrak{M}_k \models^\omega \{\varphi_0, \ldots, \varphi_k\}.$$

Let such $\mathfrak{M}_k$'s be fixed. Let $mean_k \overset{\text{def}}{=} mean^{\omega}_{\mathfrak{M}_k}$. Let $\mathfrak{A}_k \overset{\text{def}}{=} mean_k(\underset{\sim}{F}^{\omega}) \in \mathsf{Alg}_2(\mathbf{L})$. Then $\mathfrak{A}_k \in \mathsf{Alg}_1(\mathbf{L})$ also holds (cf Exercise 3.1.3). Let P^{ω} be the set of atomic formulas of $\mathcal{L}^{\omega}$. Then the function $mean_k : P^{\omega} \longrightarrow A_k$ is a valuation of the variables into $\mathfrak{A}_k$. Let $\mathcal{F}$ be a non-principal ultrafilter over ω, and let $\mathfrak{A} \overset{\text{def}}{=} \mathbf{P}_{k\in\omega}\mathfrak{A}_k/\mathcal{F}$ denote the ultraproduct of algebras $\mathfrak{A}_k$ w.r.t. $\mathcal{F}$. We define the function $v : \omega \longrightarrow A$ as follows:

$$v(i) \overset{\text{def}}{=} \langle mean_k(p_i) : k \in \omega \rangle / \mathcal{F}\,.$$

See Figure 3.1 below.

$$\begin{tikzcd}
 & \omega \arrow[dl, "mean_0"'] \arrow[dr, "mean_k"] \arrow[drrrr, "v"] & & & & \\
\mathfrak{A}_0 & \cdots & \mathfrak{A}_k & \cdots & \underset{\lim}{\Longrightarrow} & \mathfrak{A}
\end{tikzcd}$$

Figure 3.1

By assumption, $\mathfrak{M}_k \models^{\omega} \varphi_i$ for every $i \leq k$. Thus, for every $i \leq k \in \omega$, we have the following:

$$\begin{aligned}
&\mathfrak{M}_k \models^{\omega} \varphi_i \\
&\quad\Updownarrow \text{ by Definition 3.1.1 (3)(ii)} \\
&\mathfrak{M}_k \models^{\omega} \mathit{True}\nabla\varphi_i \\
&\quad\Updownarrow \text{ by Definition 3.1.1 (3)(i)} \\
&mean_k(\mathit{True}) = mean_k(\varphi_i) \\
&\quad\Updownarrow \\
&\mathfrak{A}_k \models (\varphi_i = \mathit{True})[mean_k].
\end{aligned}$$

We derived that $(\forall k \in \omega)(\forall i \leq k)\ \mathfrak{A}_k \models (\varphi_i = \mathit{True})[mean_k]$, i.e. for every $i \in \omega$, $\{k \in \omega : \mathfrak{A}_k \models \varphi_i = \mathit{True}[mean_k]\} \in \mathcal{F}$. Using Łoś' theorem, we have that

$$(\forall i \in \omega)\ \mathfrak{A} \models (\varphi_i = \mathit{True})[v]\,.$$

Since by our assumption $\mathsf{Alg}_1(\mathbf{L})$ is **Up**–closed, $\mathfrak{A} \in \mathsf{Alg}_1(\mathbf{L})$. Thus ($\exists$ cardinal $\alpha \geq \omega$) $(\exists K \subseteq M^{\alpha})\ \mathfrak{A} \cong \underset{\sim}{F}^{\alpha}/\sim_K$. Let *iso* denote this isomorphism. Let $\mathfrak{B} \overset{\text{def}}{=} \underset{\sim}{F}^{\alpha}/\sim_K$, and let $w \overset{\text{def}}{=} iso \circ v$ (i.e. w is the composition of v and *iso*). Then

$$(\forall i \in \omega)\ \mathfrak{B} \models (\varphi_i = \mathit{True})[w]$$

that is,

$$(\forall i \in \omega)\ \varphi_i[w(p_{i_0}),\dots,w(p_{i_n})]^{\mathfrak{B}} = True[w]^{\mathfrak{B}}.$$

Let P^α denote the set of atomic formulas of $\mathcal{L}^\alpha$. Let $s : P^\alpha \longrightarrow F^\alpha$ be such that for all $p \in P^\omega$ $s(p)$ is an element of the congruence class $w(p)$. For every $i \in \omega$, let $\hat{\varphi}_i \in F^\alpha$ be $\varphi_i(p_{i_0}/s(p_{i_0}),\dots,p_{i_n}/s(p_{i_n}))$, where all the atomic formulas (elements of P^ω) occurring in φ_i are among $\{p_{i_0},\dots,p_{i_n}\}$. Then for every $i \in \omega$ we have,

$$\begin{aligned} \varphi_i[s(p_{i_0})/\sim_K,\dots,s(p_{i_n})/\sim_K]^{\mathfrak{B}} &= True^{\mathfrak{B}} \\ &\Downarrow \quad (\sim_K \text{ is a congruence on } \underset{\sim}{F}^\alpha) \\ \varphi_i(s(p_{i_0}),\dots,s(p_{i_n}))/\sim_K &= True/\sim_K \\ &\Updownarrow \\ \hat{\varphi}_i/\sim_K &= True/\sim_K\ . \end{aligned} \tag{†}$$

Let $\mathfrak{M}$ be any model belonging to K. Then for every $i \in \omega$ we have $\mathfrak{M} \models^\alpha \hat{\varphi}_i$. Then, by (5) of Definition 3.1.1 (semantical substitution property),

$$(\exists \mathfrak{N}' \in M^\alpha)(\forall i \in \omega)\ mean^\alpha_{\mathfrak{N}'}(\varphi_i) = mean^\alpha_{\mathfrak{N}'}(True).$$

Since $True$ and φ_i belong to F^ω, by (3) of Def.3.2.3, there is a model $\mathfrak{N} \in M^\omega$ such that

$$\begin{gathered} (\forall i \in \omega)\ mean^\omega_{\mathfrak{N}}(\varphi_i) = mean^\alpha_{\mathfrak{N}'}(\varphi_i) \text{ and} \\ mean^\omega_{\mathfrak{N}}(True) = mean^\alpha_{\mathfrak{N}'}(True). \end{gathered}$$

Then, by Definition 3.1.1 (3),

$$(\forall i \in \omega)\ \mathfrak{N} \models^\omega \varphi_i,$$

which proves Theorem 3.2.6. ▮

Our next theorem states that the condition of Theorem 3.2.6 above is sufficient and also necessary for cons. compactness, and so for strong completeness (cf. Theorem 3.2.8 below).

THEOREM 3.2.7. *(cf.* [ANS84] *Thm.2.8)*
Assume **L** *is a strongly nice general logic. Then*

$$(\mathsf{Alg}_1(\mathbf{L}) \textit{ is } \mathbf{Up}\textit{–closed}) \iff (\mathbf{L} \textit{ is cons. compact}).$$

Proof of ($\Longrightarrow$): One can push through the proof of Theorem 3.2.6 for this case, as follows. Now we want to prove $\{\varphi_i : i \in \omega\} \not\models^\omega \psi$ from the assumption $\{\varphi_0,\dots,\varphi_k\} \not\models^\omega \psi$ for each $k \in \omega$. Change $\mathfrak{M}_k$ in the above proof such that $\mathfrak{M}_k \models^\omega \{\varphi_0,\dots,\varphi_k\}$ *and* $\mathfrak{M}_k \not\models^\omega \psi$. Drag this "$\not\models^\omega \psi$" part through the whole argument in exactly the same style as "$\models^\omega \varphi_k$" was treated in the original proof. Then in line (†) of the proof above we have $(\forall i \in \omega)\ \hat{\varphi}_i/\sim_K = True/\sim_K$ *and* $\hat{\psi}/\sim_K \neq True/\sim_K$ for some class $K \subseteq M^\alpha$. Now we cannot choose an arbitrary $\mathfrak{M} \in K$ but we can infer that there exists some $\mathfrak{M} \in K$ such that $(\forall i \in \omega)\ \mathfrak{M} \models^\alpha \varphi_i$ and $\mathfrak{M} \not\models^\alpha \psi$. Thus, again by (5) of Def. 3.1.1

and by (3) of Def.3.2.3, there is an $\mathfrak{N} \in M^\omega$ with $\mathfrak{N} \models^\omega \{\varphi_i : i \in \omega\}$ and $\mathfrak{N} \not\models^\omega \psi$, as was desired. ▮

Proof of ($\Longleftarrow$): Fix any set I and assume that for each $i \in I$ $\mathfrak{A}_i \in \mathsf{Alg}_1(\mathcal{L}^{\lambda_i})$ for some cardinal λ_i that is (by Theorem 3.2.5),

$$\mathfrak{A}_i \subseteq \mathbf{P}_{\mathfrak{M}\in K_i} mean^{\lambda_i}_{\mathfrak{M}}(\underset{\sim}{F}^{\lambda_i}).$$

Now let $\lambda \overset{\text{def}}{=} sup\{\lambda_i : i \in I\}$ and define $K \subseteq M^\lambda$ as

$$K \overset{\text{def}}{=} \left\{\mathfrak{N} \in M^\lambda \ : \ (\exists i \in I)(\exists \mathfrak{M} \in K_i)\ mean^{\lambda_i}_{\mathfrak{M}} = mean^{\lambda}_{\mathfrak{N}} \restriction \underset{\sim}{F}^{\lambda_i}\right\}.$$

Then $\mathfrak{P} \overset{\text{def}}{=} \mathbf{P}_{i\in I}\mathfrak{A}_i \subseteq \mathbf{P}_{\mathfrak{M}\in K} mean^{\lambda}_{\mathfrak{M}}(\underset{\sim}{F}^{\lambda})$ by (2) and (3) of Def.3.2.3.

Let $\alpha \overset{\text{def}}{=} |\mathfrak{P}|$, fix any bijection from the set P^α of atomic formulas of $\mathcal{L}^\alpha$ onto P (the universe of $\mathfrak{P}$) and let $h : \underset{\sim}{F}^\alpha \twoheadrightarrow \mathfrak{P}$ be its natural extension to a homomorphism *onto* $\mathfrak{P}$. For each $X \subseteq I$ define the congruence R_X of $\mathfrak{P}$ as follows.

$$R_X \overset{\text{def}}{=} \{(a,b) \in {}^2P \ : \ a \restriction X = b \restriction X\}.$$

Claim 3.2.7.1: Let h and R_X be as above. Then for any $X \subseteq I$ there is some $M_X \subseteq M^\alpha$ such that

(i) $(\forall\varphi,\psi \in F^\alpha)\ \left[(h(\varphi),h(\psi)) \in R_X \iff \varphi \sim_{M_X} \psi\right]$;
(ii) if $X \subseteq Y \subseteq I$ then $M_X \subseteq M_Y \subseteq M^\alpha$.

Proof of Claim 3.2.7.1: Recall that $\mathfrak{P} \subseteq \mathbf{P}_{\mathfrak{M}\in K} mean^{\lambda}_{\mathfrak{M}}(\underset{\sim}{F}^{\lambda})$. Fix some $X \subseteq I$. Then $\mathfrak{P}/R_X \cong \mathbf{P}_{i\in X}\mathfrak{A}_i$ obviously holds. Thus there is some $K_X \subseteq K$ such that $\mathfrak{P}/R_X \subseteq \mathbf{P}_{\mathfrak{M}\in K_X} mean^{\lambda}_{\mathfrak{M}}(\underset{\sim}{F}^{\lambda})$.

Now it can be proved (cf. Claim 3.2.5.1) that for every $\mathfrak{M} \in K_X$ there is some $\mathfrak{N} \in M^\alpha$ such that

$$(\forall p \in P^\alpha)\ mean^{\alpha}_{\mathfrak{N}}(p) = h(p)_{\mathfrak{M}},$$

where $h(p)_{\mathfrak{M}}$ denotes the $\mathfrak{M}^{\text{th}}$ member of the sequence $h(p)$.

For each $\mathfrak{M} \in K_X$ we can define a nonempty class $M(\mathfrak{M}) \subseteq M^\alpha$ as follows.

$$M(\mathfrak{M}) \overset{\text{def}}{=} \{\mathfrak{N} \in M^\alpha \ : \ (\forall p \in P^\alpha)\ mean^{\alpha}_{\mathfrak{N}}(p) = h(p)_{\mathfrak{M}}\}.$$

Let $M_X \overset{\text{def}}{=} \bigcup\{M(\mathfrak{M}) : \mathfrak{M} \in K_X\}$. Then M_X has property (ii) above by definition. It can be proved that M_X also has (i) (cf. the proof of Claim 3.2.5.2). ▮

By Claim 3.2.7.1 (i) above and by Fact 3.1.2.1, for each $X \subseteq I$ there is a set $\Gamma_X \subseteq F^\alpha$ such that

$$(\forall\varphi,\psi \in F^\alpha)\ \left[(h(\varphi),h(\psi)) \in R_X \iff \varphi \sim_{\Gamma_X} \psi\right].$$

Moreover, by (ii) of the above claim, for any $X, Y \subseteq I$,

$$(*) \qquad X \subseteq Y \implies \Gamma_Y \subseteq \Gamma_X.$$

Claim 3.2.7.2: Let $\mathcal{F}$ be any filter on I and let $\Gamma \overset{\text{def}}{=} \bigcup\{\Gamma_X : X \in \mathcal{F}\}$. Then for every $\varphi, \psi \in F^\alpha$

$$\varphi \sim_\Gamma \psi \quad\Longleftrightarrow\quad (\exists X \in \mathcal{F})\, \varphi \sim_{\Gamma_X} \psi.$$

Proof of Claim 3.2.7.2:

First, assume that $(\exists X \in \mathcal{F})\, \varphi \sim_{\Gamma_X} \psi$. Then, since $\Gamma_X \subseteq \Gamma$, $\varphi \sim_\Gamma \psi$ obviously holds.

On the other hand, assume $\varphi \sim_\Gamma \psi$. Then $\Gamma \models^\alpha \varphi\nabla\psi$. Then, by the cons. compactness of $\mathcal{L}^\alpha$, there is some $\Delta \subseteq_\omega \Gamma$ with $\Delta \models^\alpha \varphi\nabla\psi$. Say, $\Delta = \{\chi_0, \ldots, \chi_{n-1}\}$. Since $\Delta \subseteq \Gamma$, $(\forall j < n)(\exists X_j \in \mathcal{F})\ \chi_j \in \Gamma_{X_j}$. Let $X \overset{\text{def}}{=} \bigcap\{X_j : j < n\}$. Then $X \in \mathcal{F}$, since $\mathcal{F}$ is a filter. Now $\Delta \subseteq \Gamma_{X_0} \cup \cdots \cup \Gamma_{X_{n-1}} \subseteq \Gamma_X$ holds by $(*)$ above, thus $\Gamma_X \models^\alpha \varphi\nabla\psi$, which implies $\varphi \sim_{\Gamma_X} \psi$. ∎

Now we want to prove that $\mathfrak{P}/\mathcal{F} \in \mathsf{Alg}_1(\mathbf{L})$. We show that $\mathfrak{P}/\mathcal{F} \cong \underset{\sim}{F}^\alpha/\sim_\Gamma$. That is,

$$(\forall\varphi, \psi \in F^\alpha)\ \big[h(\varphi) \sim_{\mathcal{F}} h(\psi) \quad\Longleftrightarrow\quad \varphi \sim_\Gamma \psi\big]$$

holds. Indeed,

$$\begin{aligned} & h(\varphi) \sim_{\mathcal{F}} h(\psi) \\ \Longleftrightarrow\quad & (\exists X \in \mathcal{F})\ (h(\varphi), h(\psi)) \in R_X \\ \overset{\text{Claim 3.2.7.1 (i)}}{\Longleftrightarrow}\quad & (\exists X \in \mathcal{F})\ \varphi \sim_{\Gamma_X} \psi \\ \overset{\text{Claim 3.2.7.2}}{\Longleftrightarrow}\quad & \varphi \sim_\Gamma \psi, \end{aligned}$$

which completes the proof of Theorem 3.2.7. We note that we proved that $\mathsf{Alg}_1(\mathbf{L})$ is closed under taking arbitrary *reduced* products (not only ultraproducts). ∎

THEOREM 3.2.8:

Assume $\mathbf{L} = \{\mathcal{L}^\alpha : \alpha \text{ is a cardinal}\}$ is strongly nice general logic. Then

$$\mathsf{Alg}_1(\mathbf{L}) \text{ is a finitely axiomatizable quasivariety}$$
$$\Longleftrightarrow$$
$$(\exists \text{ Hilbert–style } \vdash)(\forall \text{ cardinal } \alpha)(\vdash \text{ is strongly complete and strongly sound for } \mathcal{L}^\alpha).$$

To prove Theorem 3.2.8 we need the following claim.

Claim 3.2.8.1: For every cardinal $\alpha \geq \omega$ and for every quasiequation q

$$\mathsf{Alg}_2(\mathcal{L}^\alpha) \models q \implies \mathsf{Alg}_2(\mathbf{L}) \models q.$$

Proof of Claim 3.2.8.1: Fix a cardinal α and a quasiequation q with $\mathsf{Alg}_2(\mathcal{L}^\alpha) \models q$. Let $\mathfrak{A} \in \mathsf{Alg}_2(\mathcal{L}^\alpha)$ for some cardinal α. Then there is some $\mathfrak{M} \in M^\alpha$ with $\mathfrak{A} = mean^\alpha_{\mathfrak{M}}(\underset{\sim}{F}^\alpha)$.

First assume that $\alpha \leq \alpha$. [illegible] (3) of Definition 3.2.3, there is an $(\mathfrak{N} \in M^\alpha)$ $mean^\alpha_{\mathfrak{N}} \restriction \underset{\sim}{F}^\alpha = mean^\alpha_{\mathfrak{M}}$. Then $\mathfrak{A} \subseteq mean^\alpha_{\mathfrak{N}}(\underset{\sim}{F}^\alpha) \in \mathsf{Alg}_2(\mathcal{L}^\alpha)$, thus $\mathfrak{A} \models q$, since quasiequations are preserved under taking subalgebras.

Now let $\alpha \geq \alpha$ and assume that $\mathfrak{A} \not\models q[k]$ for some evaluation k of the variables. Say, let $k(x_i) \overset{\text{def}}{=} mean^{\alpha}_{\mathfrak{M}}(\gamma_i))$ $(1 \leq i \leq n)$, assuming that $x_1, \ldots, x_n$ are the only variables occurring free in q. Assume that the atomic formulas occurring in the formulas $\gamma_1, \ldots, \gamma_n$ are among $p_{i_1}, \ldots, p_{i_m}$ and let s be the following substitution:

$$(\forall 1 \leq j \leq m)\ s(p_j) \overset{\text{def}}{=} p_{i_j}.$$

Then, by (5) of Definition 3.1.1,

$$(\exists \mathfrak{N} \in M^{\alpha})(\forall 1 \leq i \leq n)\ mean^{\alpha}_{\mathfrak{M}}(\gamma_i) = mean^{\alpha}_{\mathfrak{N}}(\gamma_i(p_{i_1}/p_1, \ldots, p_{i_m}/p_m)).$$

By (3) of Definition 3.2.3, $(\exists \mathfrak{N}' \in M^{\alpha})\ mean^{\alpha}_{\mathfrak{N}} \restriction \underset{\sim}{F}^{\alpha} = mean^{\alpha}_{\mathfrak{N}'}$. Now, let $\mathfrak{B} \overset{\text{def}}{=} mean^{\alpha}_{\mathfrak{N}'}$ and let $k'(x_i) \overset{\text{def}}{=} mean^{\alpha}_{\mathfrak{N}'}(\gamma_i(p_1, \ldots, p_m))$. Then $\mathfrak{A} \not\models q[k]$ implies $\mathfrak{B} \not\models q[k']$, which contradicts to $\mathfrak{B} \in \mathsf{Alg}_2(\mathcal{L}^{\alpha})$. ▮

Proof of ($\Longrightarrow$) of Theorem 3.2.8: Assume that Ax is a finite set of quasiequations axiomatizing $\mathsf{Alg}_2(\mathbf{L})$. Since $\mathsf{Alg}_1(\mathbf{L}) = \mathbf{SP}\mathsf{Alg}_2(\mathbf{L})$ (cf. Theorem 3.2.5), by Claim 3.2.8.1 above, Ax also axiomatizes the quasivariety generated by $\mathsf{Alg}_2(\mathcal{L}^{\alpha})$ for each infinite cardinal α. Thus, by Theorem 3.2.1, for each $\alpha \geq \omega$ there is a finitely complete and strongly sound Hilbert-style inference system $\vdash$ for $\mathcal{L}^{\alpha}$. Moreover, checking the proof of Theorem 3.2.1 one can observe that the same inference system $\vdash$ works for every $\alpha \geq \omega$. We show that for any cardinal λ, $\vdash$ is strongly complete for $\mathcal{L}^{\lambda}$. Assume that for some $\Gamma \cup \{\varphi\} \subseteq F^{\lambda}$ $\Gamma \models^{\lambda} \varphi$. Then there is some $\alpha \geq \omega$ such that $\Gamma \cup \{\varphi\} \subseteq F^{\alpha}$ and $\Gamma \models^{\alpha} \varphi$ (cf. Remark 3.2.3.1 above). Since quasivarieties are Up–closed, $\mathcal{L}^{\alpha}$ is cons. compact by Theorem 3.2.7. Therefore there is a finite subset Σ of Γ such that $\Sigma \models^{\alpha} \varphi$. Thus, by finite completeness $\Sigma \vdash \varphi$, which implies $\Gamma \vdash \varphi$ by the definition of derivability (Def.3.1.5). ▮

Proof of ($\Longleftarrow$) of Theorem 3.2.8: If $\vdash$ is strongly complete then it is also finitely complete. Thus, by Theorem 3.2.1, the quasivariety generated by $\mathsf{Alg}_2(\mathcal{L}^{\alpha})$ is finitely axiomatizable for each cardinal α. On the other hand, strong completeness implies cons. compactness, as follows. Assume that for some $\Gamma \cup \{\varphi\} \subseteq F^{\alpha}$ $\Gamma \models^{\alpha} \varphi$. Then $\Gamma \vdash \varphi$, which implies by Definition 3.1.5 that there is a finite subset Σ of Γ such that $\Sigma \vdash \varphi$. Then, by soundness, $\Sigma \models^{\alpha} \varphi$. Now, by Theorem 3.2.7, $\mathsf{Alg}_1(\mathbf{L})$ is Up–closed. But by Theorem 3.2.5, it is also closed under $\mathbf{S}$ and $\mathbf{P}$, thus it is a quasivariety. This and the fact that the quasivarieties generated by $\mathsf{Alg}_2(\mathcal{L}^{\alpha})$ are finitely axiomatizable (with the same set Ax of quasiequations, as the proof of Theorem 3.2.1 shows) imply that $\mathsf{Alg}_1(\mathbf{L})$ is a finitely axiomatizable quasivariety. ▮

EXERCISE 3.2.4: Show that $\mathcal{L}_S$ and $S5$ have strongly complete and sound Hilbert-style inference systems. Give such calculi. (Hint: Use that the corresponding classes of algebras ($\mathsf{Alg}_2(\mathbf{L}_S) = \mathsf{BA}$ and $\mathsf{Alg}_2(\mathbf{L}_{S5}) = \mathsf{Cs}_1$) generate finitely axiomatizable varieties.) ◀

3.3. Some universal algebraic tools for algebraic logic

So far we have seen that the algebraic counterparts $\mathsf{Alg}_1(\mathbf{L})$ of many logics are quasivarieties. However, there are logics for which $\mathsf{Alg}_1(\mathbf{L})$ is nicer, it is a variety (that is, $\mathsf{Alg}_1(\mathcal{L})$ is closed not only under $\mathbf{S}$ and $\mathbf{P}$ but also $\mathbf{H}$). Usually, it is a difficult task to prove that a certain class of algebras is closed under homomorphism. Theorem 3.3.1 below gives us considerable help by proving that certain quasivarieties are already varieties.

DEFINITION 3.3.1:

(i) A class K of algebras is said to *have a discriminator term* iff there is a term $\tau(x,y,z,u)$ in the language of K such that in every member of K we have

$$\tau(x,y,z,u)=\begin{cases} z, & \text{if } x=y,\\ u, & \text{otherwise.}\end{cases}$$

(ii) A variety V is called a *discriminator variety* if the class $Sir(V)$ of subdirectly irreducible members of V has a discriminator term. ◀

EXERCISES 3.3.1:

(1) Show that if K has a discriminator term then K consists of simple algebras.

(2) Assume that the Boolean operations $-,\wedge,0,1$ are available in K and that they satisfy the Boolean axioms (i.e. every element of K is a Boolean algebra with some further operations). This property will be referred to as 'K *has a Boolean reduct*'. Prove that K has a discriminator term iff there is a term $c(x)$ in the language of K such that

$$c(x)=\begin{cases} 0, & \text{if } x=0,\\ 1, & \text{otherwise}\end{cases}$$

in every member of K. (Hint: $\tau(x,y,z,u)=\big[c(x\oplus y)\wedge u\big]\vee\big[z\wedge -c(x\oplus y)\big]$. Here $\oplus$ denotes symmetric difference.)

(3) Check how much simplification one can achieve in the proof of Thm.1.3.3.1 below under assuming that K has a Boolean reduct (cf. item (2) above). ◀

THEOREM 3.3.1. *Let K be a class of similar algebras. Assume that K has a discriminator term. Then*

$$\mathbf{HSP}\ \mathsf{K}=\mathbf{SPUp}\ \mathsf{K}.$$

To prove Theorem 3.3.1 we need the following lemmas.

Lemma 3.3.1.1: Assume that the class K of algebras has a discriminator term. Let I be a set and $\{\mathfrak{A}_i : i\in I\}\subseteq\mathsf{K}$. Let $\mathfrak{A}\subseteq\mathbf{P}_{i\in I}\mathfrak{A}_i$ and let $\theta\in Con(\mathfrak{A})$. For any $a,b\in A$, let $Eq(a,b)\overset{\mathrm{def}}{=}\{i\in I : a_i=b_i\}$. Then

(1) $(\forall\langle a,b\rangle,\langle c,d\rangle\in\theta)(\exists\langle e,f\rangle\in\theta)\ Eq(e,f)=Eq(a,b)\cap Eq(c,d)$.

(2) $(\forall a,b,c,d\in A)\big[(\langle a,b\rangle\in\theta\ \ \&\ \ Eq(a,b)\subseteq Eq(c,d))\Longrightarrow\langle c,d\rangle\in\theta\big]$.

Proof of Lemma 3.3.1.1: Let τ be a discriminator term on K.

Let $\langle a,b\rangle, \langle c,d\rangle \in \theta$, and let $e = \tau(a,b,c,a)$, $f = \tau(a,b,d,b)$. Then $\langle e,f\rangle \in \theta$.

Assume $i \notin Eq(a,b) \cap Eq(c,d)$. If $i \notin Eq(a,b)$, then $e_i = a_i \neq b_i = f_i$. If $i \in Eq(a,b)$, then $i \notin Eq(c,d)$, thus $e_i = c_i \neq d_i = f_i$. Thus $i \notin Eq(e,f)$, proving $Eq(e,f) \subseteq Eq(a,b) \cap Eq(c,d)$.

Assume $i \in Eq(a,b) \cap Eq(c,d)$. Then $e_i = c_i = d_i = f_i$, thus $i \in Eq(e,f)$, proving $Eq(a,b) \cap Eq(c,d) \subseteq Eq(e,f)$. By this we have proved (1).

To see (2), assume $\langle a,b\rangle \in \theta$ and $Eq(a,b) \subseteq Eq(c,d)$. By $\langle a,b\rangle \in \theta$, we have $c = \tau(a,a,c,d)\theta\tau(a,b,c,d) = x$. We will show that $x = d$. If $a_i = b_i$, then $c_i = d_i$ by $Eq(a,b) \subseteq Eq(c,d)$, hence $x_i = d_i$. If $a_i \neq b_i$, then $x_i = d_i$ by the choice of τ. Thus $\langle c,d\rangle \in \theta$, proving (2). ∎

Lemma 3.3.1.2: Let $\mathsf{K}, I, \mathfrak{A}_i, \mathfrak{A}, \theta$ be as in the formulation of Lemma 3.3.1.1. Then there is a filter $\mathcal{F}$ over I such that

$$(*) \qquad (\forall a,b \in A)(\langle a,b\rangle \in \theta \iff Eq(a,b) \in \mathcal{F}).$$

Proof of Lemma 3.3.1.2: Let $\mathsf{K}, I, \mathfrak{A}_i, \mathfrak{A}, \theta, \tau$ be as above. Let

$$\mathcal{F} \overset{\text{def}}{=} \{X \subseteq I : X \supseteq Eq(a,b) \text{ for some } \langle a,b\rangle \in \theta\}.$$

We show that $\mathcal{F}$ is a filter over I, as follows.

$\mathcal{F}$ is closed under finite intersections: $X, Y \in \mathcal{F} \Longrightarrow X \cap Y \supseteq Eq(a,b) \cap Eq(c,d)$ for some $\langle a,b\rangle, \langle c,d\rangle \in \theta$. Then $Eq(a,b) \cap Eq(c,d) = Eq(e,f)$ for some $\langle e,f\rangle \in \theta$ by Lemma 3.3.1.1 (1). Thus $X \cap Y \supseteq Eq(e,f)$, for $\langle e,f\rangle \in \theta$.

From the definition of $\mathcal{F}$ it follows that $I \in \mathcal{F}$ and that

$$(\forall Y \subseteq I)(\exists X \in \mathcal{F})[Y \supseteq X \Longrightarrow Y \in \mathcal{F}].$$

We have seen that $\mathcal{F}$ is a filter over I. It remains to show that $\mathcal{F}$ satisfies $(*)$ above. $Eq(a,b) \in \mathcal{F} \iff (\exists\langle c,d\rangle \in \theta)\ Eq(c,d) \subseteq Eq(a,b)$ so, by Lemma 3.3.1.1 (2), $\langle a,b\rangle \in \theta$, proving $Eq(a,b) \in \mathcal{F} \Longrightarrow \langle a,b\rangle \in \theta$. The other direction follows from the definition of $\mathcal{F}$. ∎

Recall that for an arbitrary class K of algebras,

$$\mathbf{P}^{\mathbf{r}}\mathsf{K} \overset{\text{def}}{=} \mathbf{I}\,\{\mathbf{P}_{i\in I}\mathfrak{A}_i/\mathcal{F} : \mathcal{F} \text{ is a filter over the set } I, \text{ and } (\forall i \in I)\mathfrak{A}_i \in \mathsf{K}\}\ .$$

The following is an easy fact of elementary universal algebra (cf. also e.g. Burris–Sankappanavar [BS81] or Németi–Sain [NS81]).

Lemma 3.3.1.3: Let K be an arbitrary class of similar algebras. Then

$$\mathbf{P}^{\mathbf{r}}\mathsf{K} \subseteq \mathbf{SPUp}\ \mathsf{K}.$$

Proof of Lemma 3.3.1.3: Let I be a set, $\{\mathfrak{A}_i : i \in I\} \subseteq \mathsf{K}$, $\mathcal{F}$ a filter over I, $\mathfrak{A} \overset{\text{def}}{=} \mathbf{P}_{i\in I}\mathfrak{A}_i/\mathcal{F} \in \mathbf{P}^{\mathbf{r}}\mathsf{K}$. Let

$$U \overset{\text{def}}{=} \{\mathcal{G} : \mathcal{G} \text{ an ultrafilter over } I \text{ and } \mathcal{G} \supseteq \mathcal{F}\}.$$

Let $h : \mathfrak{A} \longrightarrow \mathbf{P}\langle \mathbf{P}_{i\in I}\mathfrak{A}_i/\mathcal{G} : \mathcal{G} \in U\rangle$ be defined as $h(a/\mathcal{F}) \overset{\text{def}}{=} \langle a/\mathcal{G} : \mathcal{G} \in U\rangle$. It is not hard to check that h is an embedding, therefore $\mathfrak{A} \in \mathbf{SPUp}\ \mathsf{K}$. ▮

Proof of Theorem 3.3.1: Let $\mathfrak{B} \in \mathbf{HSP}\ \mathsf{K}$ be arbitrary. Then there are $I, \mathfrak{A}_i, \theta, \mathfrak{A}$ as in the formulation of Lemma 3.3.1.1 such that $\mathfrak{B} = \mathfrak{A}/\theta$. By Lemma 3.3.1.2, there is a filter $\mathcal{F}$ on I such that $\mathfrak{A}/\theta \subseteq \mathbf{P}_{i\in I}\mathfrak{A}_i/\mathcal{F}$, thus $\mathfrak{B} \in \mathbf{SP^r}\ \mathsf{K}$. This shows $\mathbf{HSP}\ \mathsf{K} \subseteq \mathbf{SP^r}\ \mathsf{K}$. By Lemma 3.3.1.3, $\mathbf{SP^r}\ \mathsf{K} \subseteq \mathbf{SSPUp}\ \mathsf{K} = \mathbf{SPUp}\ \mathsf{K}$, thus $\mathbf{HSP}\ \mathsf{K} \subseteq \mathbf{SPUp}\ \mathsf{K}$.

On the other hand, $\mathbf{SPUp}\ \mathsf{K} \subseteq \mathbf{HSP}\ \mathsf{K}$, by $\mathbf{Up} \subseteq \mathbf{HP}$, $\mathbf{PH} \subseteq \mathbf{HP}$, $\mathbf{SH} \subseteq \mathbf{HS}$, and $\mathbf{PP} = \mathbf{P}$. Thus we completed the proof of Theorem 3.3.1. ▮

COROLLARY 3.3.1: Assume K has a discriminator term. Then

(i) K is contained in some discriminator variety.

(ii) The subdirectly irreducible members of $\mathbf{HSP}\ \mathsf{K}$ are exactly the subdirectly irreducibles of $\mathbf{SUp}\ \mathsf{K}$.

Proof:

(ii): Let $\mathfrak{A}$ be a subdirectly irreducible member of $\mathbf{HSP}\ \mathsf{K}$. By Theorem 3.3.1, $\mathfrak{A} \in \mathbf{SP}(\mathbf{SUp}\ \mathsf{K})$. Then $\mathfrak{A}$ is a subdirect product of algebras from $\mathbf{SUp}\ \mathsf{K}$. By irreducibility, then $\mathfrak{A} \in \mathbf{SUp}\ \mathsf{K}$. This proves (ii).

(i): The discriminator term τ which works for K also works for $\mathbf{SUp}\ \mathsf{K}$, since the discriminator property

$$\forall x, y, z, u\ \big([x \neq y \Rightarrow \tau(x, y, z, u) = u] \wedge [x = y \Rightarrow \tau(x, y, z, u) = z]\big)$$

is defined by a universal formula, thus is preserved under $\mathbf{SUp}$. Thus $\mathbf{SUp}\ \mathsf{K}$ has a discriminator term. But by (ii) the class $Sir(\mathbf{HSP}\ \mathsf{K})$ of subdirectly irreducibles of $\mathbf{HSP}\ \mathsf{K}$ is in $\mathbf{SUp}\ \mathsf{K}$. Then by definition, $\mathbf{HSP}\ \mathsf{K}$ is a discriminator variety. ▮

3.4. Distinguished Logics

In this section we give a brief summary of the logics defined so far and give some further ones. Let P be an arbitrary but fixed set of atomic formulas. For each of the logics in this section, the class of models (corresponding to P) will be a subclass of the following one:

$$\mathsf{Mod}_0 \overset{\text{def}}{=} \{\langle W, v\rangle : W \text{ is a set and } v : P \longrightarrow \mathcal{P}(W) \text{ is a function}\}\ .$$

In all our logics we will have the Boolean logical connectives and some extra-Boolean logical connectives. According to a rather respectable (and useful) tradition an extra-Boolean connective is called a *modality* iff it distributes over disjuction. This will not be true for all of our connectives (Homework: check which ones). Thus, regrettably we sometimes ignore this useful tradition. For this tradition cf. e.g. Venema [V92] Appendix A (pp. 143–152). When specifying a logic $\mathcal{L}$, we will discuss *only* its extra-Booleans, since the Booleans are standard. For a logic $\mathcal{L}$, $\mathsf{Mod}(\mathcal{L})$ is the class of models of $\mathcal{L}$. For $w \in W$, $w \Vdash \varphi$ means that φ is true at w.

(1) $\mathcal{L}_S$: *propositional logic* (cf. Def.2.3). $\mathsf{Mod}(\mathcal{L}_S) \overset{\text{def}}{=} \mathsf{Mod}_0$.

(2) $S5$: *Modal logic $S5$* (cf. Def.2.4). $\mathsf{Mod}(S5) \overset{\text{def}}{=} \mathsf{Mod}_0$. Extra-Boolean: $\Diamond$. Its meaning is

$$w \Vdash \Diamond\varphi \iff (\exists w' \in W)\ w' \Vdash \varphi\,.$$

(3) D: *Difference logic* or *"Some-other-time logic"*. $\mathsf{Mod}(D) \overset{\text{def}}{=} \mathsf{Mod}_0$. Extra-Boolean: D. Its meaning is

$$w \Vdash D\varphi \iff (\exists w' \in W \smallsetminus \{w\})\ w' \Vdash \varphi\,.$$

(4) $\Diamond_\kappa$: *κ-times logic*. Here κ is any fixed *cardinal* (may be infinite). $\mathsf{Mod}(\Diamond_\kappa) \overset{\text{def}}{=} \mathsf{Mod}_0$. Extra-Boolean: $\Diamond_\kappa$. Its meaning is

$$w \Vdash \Diamond_\kappa\varphi \iff (\exists H \subseteq W)(|H| = \kappa \quad \& \quad (\forall w' \in H)\ w' \Vdash \varphi)\,.$$

(5) Tw and $\Diamond_n$: *Twice logic* and *n-times logic*. Here $Tw \overset{\text{def}}{=} \Diamond_2$ and $\Diamond_n$ is $\Diamond_\kappa$ for $\kappa = n < \omega$.

(6) $\mathcal{L}_{\mathrm{PAIR}}$.

$$\mathsf{Mod}(\mathcal{L}_{\mathrm{PAIR}}) \overset{\text{def}}{=} \{\langle W, v\rangle \in \mathsf{Mod}_0 : W \subseteq U \times U \text{ for some set } U\}\ .$$

Extra-Boolean: $\circ$ (binary). Its meaning is

$$\langle ab\rangle \Vdash \varphi \circ \psi \iff \exists c(\langle ac\rangle, \langle cb\rangle \in W \quad \& \quad \langle ac\rangle \Vdash \varphi \quad \& \quad \langle cb\rangle \Vdash \psi)\,.$$

(7) $\mathcal{L}_{\mathrm{REL}}$.

$$\mathsf{Mod}(\mathcal{L}_{\mathrm{REL}}) \overset{\text{def}}{=} \{\langle W, v\rangle \in \mathsf{Mod}_0 : W = U \times U \text{ for some set } U\}\ .$$

The extra–Boolean and its meaning is same as in $\mathcal{L}_{\mathrm{PAIR}}$.

(8) $\mathcal{L}_{\mathrm{ARROW}}$: van Benthem's *arrow logic*. $\mathrm{Mod}(\mathcal{L}_{\mathrm{ARROW}}) \overset{\mathrm{def}}{=} \mathrm{Mod}(\mathcal{L}_{\mathrm{PAIR}})$. Extra–Booleans: $\circ$, $^{-1}$, Id. Meaning of $\circ$ is the same as in $\mathcal{L}_{\mathrm{PAIR}}$.

$$(\langle ab\rangle \Vdash \varphi^{-1} \iff (\langle ba\rangle \in W \text{ and } \langle ba\rangle \Vdash \varphi)), \quad (\langle ab\rangle \Vdash Id \iff a = b)\,.$$

(9) $\mathcal{L}_{\mathrm{RA}}$: restriction of $\mathcal{L}_{\mathrm{ARROW}}$ to the models of $\mathcal{L}_{\mathrm{REL}}$. $\mathrm{Mod}(\mathcal{L}_{\mathrm{RA}}) \overset{\mathrm{def}}{=} \mathrm{Mod}(\mathcal{L}_{\mathrm{REL}})$. Extra–Booleans and their meanings are exactly as in $\mathcal{L}_{\mathrm{ARROW}}$.

(10) For $\mathcal{L}_n$ cf. Def.2.5. But it is important to note that $\mathcal{L}_n$ could be defined as

$$\mathrm{Mod}(\mathcal{L}_n) \overset{\mathrm{def}}{=} \{\langle W, v\rangle \in \mathrm{Mod}_0 : W = {}^nU \text{ for some set } U\}\,.$$

The extra–Booleans are "$\exists v_i$" and "$v_i = v_j$" for $i, j < n$.

SUMMARY:

$\mathcal{L}_S$	propositional logic
$S5$	modal logic, where the accessibility relation is $W \times W$ for a set W of "possible worlds"
D	difference logic (or "some–other–time" logic)
$\Diamond_2$ or Tw	twice logic
$\Diamond_n$	n–times logic ($n \in \omega$)
$\Diamond_\kappa$	κ–times logic (κ is any cardinal)
$\mathcal{L}_{\mathrm{PAIR}}$	set of worlds is arbitrary $W \subseteq U \times U$ for some U, only extra–Boolean is $\circ^W$
$\mathcal{L}_{\mathrm{REL}}$	set of worlds is $U \times U$ for some U, only extra–Boolean is $\circ$
$\mathcal{L}_{\mathrm{RA}}$	(logic of relation algebras) set of worlds is $U \times U$, extra–Booleans $\circ$, $^{-1}$, Id
$\mathcal{L}_{\mathrm{ARROW}}$	van Benthem's arrow logic. Set W of worlds is as in $\mathcal{L}_{\mathrm{PAIR}}$, extra–Booleans are as in $\mathcal{L}_{\mathrm{RA}}$, but now relativized to W
$\mathcal{L}_n$	first–order logic restricted to the first n variables ($n \in \omega$)
$\mathcal{L}_{\omega\omega}$	(usual) first–order logic with ω many variables

Let

$$\mathbf{L} \overset{\mathrm{def}}{=} \{\mathcal{L}_S, S5, D, Tw, \Diamond_n, \Diamond_\kappa, \mathcal{L}_{\mathrm{PAIR}}, \mathcal{L}_{\mathrm{REL}}, \mathcal{L}_{\mathrm{RA}}, \mathcal{L}_{\mathrm{ARROW}}, L_n, L_{\omega\omega} : n \in \omega, \kappa \in \mathit{Card}\}\,.$$

DISTINGUISHED PROPERTIES to be checked for every $\mathcal{L} \in \mathbf{L}$:

(The reason for looking at these properties is that they distinguish first–order like logics from propositional like logics.)

dec The set of all valid formulas of $\mathcal{L}$ is decidable. (Briefly: $\mathcal{L}$ is decidable.)

fmp $\mathcal{L}$ has the finite model property (fmp).

$$\mathcal{L} \textit{ has the fmp} \overset{\mathrm{def}}{\iff} (\forall\varphi \in F_{\mathcal{L}})[\models_{\mathcal{L}} \varphi \iff (\forall\mathfrak{M} \in M_{\mathcal{L}})(|\mathfrak{M}| < \omega \Rightarrow \models_{\mathcal{L}} \varphi)].$$

r.e. The set of all valid formulas of $\mathcal{L}$ is recursively enumerable (r.e.). (Briefly: $\mathcal{L}$ is r.e.)

Remark: If $\mathcal{L}$ is r.e. and $\mathcal{L}$ has the fmp the $\mathcal{L}$ is decidable.

fax $\mathsf{Alg}_1(\mathcal{L})$ is finitely axiomatizable (fax).

Gip $\mathcal{L}$ has Gödel's incompleteness property (Gip).

$\mathcal{L}$ *has Gip* $\overset{\text{def}}{\iff}$ there is a finitely axiomatizable set T of formulas of $\mathcal{L}$ such that every consistent extension of T is undecidable. That is,
$(\exists\varphi \in F)(\forall T \subseteq F)[(\varphi \in T \& T \text{ is consistent}) \Rightarrow (\{\psi : T \models \psi\} \text{ is undecidable})]$.

clm The distinction between set–models and class–models counts (clm). That is: Assume $|P| < \omega$. (P is the set of atomic formulas of $\mathcal{L}$. E.g., P is the set of propositional variables in cases of $\mathcal{L}_S$ or $S5$ or D; and it is the set of relation symbols [similarity type] in cases of $\mathcal{L}_n$ or $\mathcal{L}_{\omega\omega}$.)

We say that *clm in the logic* $\mathcal{L}$ $\overset{\text{def}}{\iff}$ ($\exists$ class–model $\mathfrak{M}$)
[$Th(\mathfrak{M})$ is not a class (hence is not a set either, i.e., does not exist)].

unm Again assume $|P| < \omega$. $(\exists\mathfrak{M} \in M_{\mathcal{L}})[Th(\mathfrak{M})$ is undecidable].

COMPARISON OF LOGICS IN L: (An arrow points to the place where the property in question becomes true "moving from left to right". Hence in principle it should always point to a gap between two logics.)

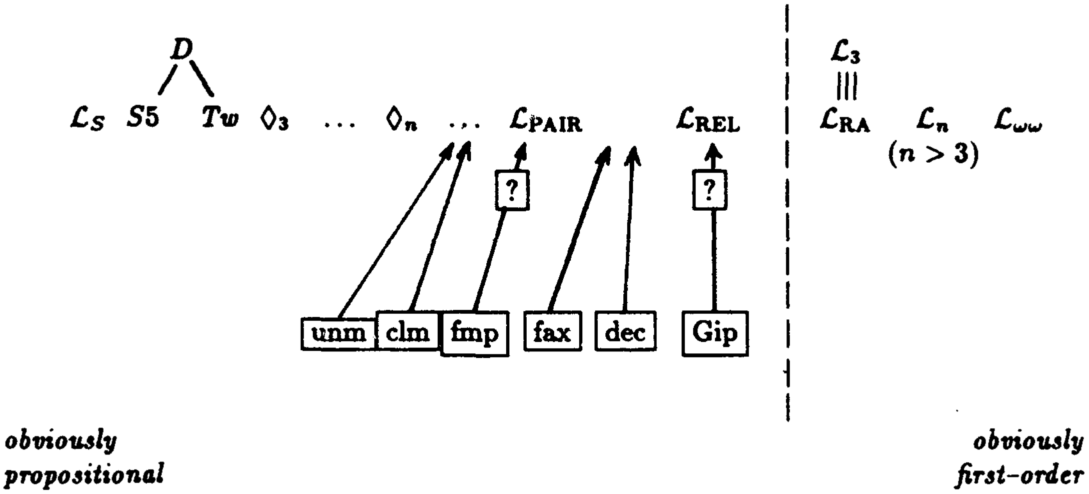


Figure 3.2

EXERCISES 3.4.1:

(1) Write up a detailed definition of $\mathcal{L}_n$ as a modal logic following the hint given in item (10) above.

(2) (*Important!*) Show that all the logics introduced above are *nice* logics. It is especially important to do for $\mathcal{L}_n$! (For $\mathcal{L}_{\omega\omega}$ it is hard, needs a reformulation of $\mathcal{L}_{\omega\omega}$ and was done e.g. in Blok–Pigozzi [**BP89**]. C.f. also Simon [**Si91**] and the references therein. It is recommended *not* to do this exercise for $\mathcal{L}_{\omega\omega}$ at this point.)

(3) Check which claims represented on Figure 3.2 were proved in the text. Try to prove the missing ones. ◄

References

[AH91] Anellis,H. and Houser,N., *The nineteenth century roots of universal algebra and algebraic logic: A critical-bibliographical guide for the contemporary logician*, In: Algebraic Logic (Proc. Conf. Budapest 1988) Colloq. Math. Soc. J. Bolyai Vol 54, North-Holland, Amsterdam (1991), 1-36.

[ANS84] Andréka,H. Németi,I. and Sain,I., *Abstract model theoretic approach to algebraic logic*, Preprint (1984), updated in 1988, 1992, 70pp.

[BP89] Blok,W.J. and Pigozzi,D., *Algebraizable logics*, Memoirs Amer. Math. Soc. Vol 77,396 (1989), vi+78 pp.

[BS81] Burris,S. and Sankappanavar,H.P., *A course in universal algebra*, Graduate Texts in Mathematics, Springer-Verlag, New York (1981).

[HMT85] Henkin,L. Monk,J.D. and Tarski,A., *Cylindric Algebras Part I, Part II*, North-Holland, Amsterdam (1985).

[Ma91] Maddux,R., *The origin* ..., Studia Logica Vol L, No 3/4 (1991), 421–456 pp.

[M76] Monk,J.D., "Mathematical Logic," Springer-Verlag, 1976.

[N91] Németi,I., *Algebraization of quantifier logics, an introductory overview*, Studia Logica Vol 50, No 3/4 (a special issue devoted to Algebraic Logic, eds.: W. J. Blok and D. L. Pigozzi) (1991), 485-570. (Strongly updated and expanded [e.g. with proofs] version is available from author.)

[P89] "Possible worlds in Humanities, Arts and Sciences," W.de Gruyer, Berlin–New York, 1989, pp. 450 p.

[SN81] Sain,I. and Németi,I., *Cone–implicational subcategories and some Birkhoff–type theorems*, In: Universal Algebra (Proc. Conf. Esztergom Hungary 1977) Colloq. Math. Soc. J. Bolyai — North–Holland Vol 29 (1981), 535–578.

[S80/a] Sain,I., *Dogmas on Language*, Manuscript (in Hungarian), 1980.

[S80/b] Sain,I., *Cognition, Learning, (Rats and) Logic*, Manuscript (in Hungarian), 1980.

[Si91] Simon,A., *Finite Schema Completeness for Typeless Logic and Representable Cylindric Algebras*, Algebraic Logic (Proc. Conf. Budapest 1988) Colloq. Math. Soc. J. Bolyai Vol 54, North-Holland, Amsterdam, 665–670.

[Si92] Simon,A., *What the Finitization Problem is Not?*, Banach Algebraic Logic Conference, to appear.

[V92] Venema,Y., *Many-Dimensional Modal Logic*, Ph.D. Dissertation, Institute for Logic, Language and Computation, Univ. of Amsterdam.

RELATION ALGEBRAS FOR REASONING ABOUT TIME, SPACE, AND PROGRAMS

ROGER D. MADDUX

May 11, 1993

ABSTRACT. This paper presents a brief survey of relation algebras and the calculus of relations, followed by two examples of their use in computer science: constraint satisfaction problems for relation algebras and a relational model for Dijkstra's axiomatic semantics for computer programs (centered on the predicate transformers called "weakest precondition" and "weakest liberal precondition"). The former topic is illustrated by the "interval algebra", a relation algebra which arose from Allen's work on temporal reasoning, and by "compass algebras", which are designed for similar reasoning about space. It will be shown here that the constraint satisfiability problem is NP-complete for almost all compass and interval algebras.

1. THE CALCULUS OF RELATIONS AND RELATION ALGEBRAS

Composition of binary relations was introduced to logic by Augustus De Morgan [34], [35] (see [36], pp. 55–57, 208, 221, *etc.*). De Morgan observed that the syllogism "every A is a B, every B is a C, so every A is a C" remains valid if the copula "is" is replaced by any transitive relation L. De Morgan went further, noting that if LM is the composition of the relation L with the relation M, that is, A is an LM of B just in case A is an L of an M of B, then the following syllogism is valid: "if every A is an L of a B, and every B is an M of a C, then every A is an LM of a C." De Morgan [35] (see [36], p. 222) denoted the converse of the relation L by L^{-1} and its contrary by not-L, and observed that these operations commute: the converse of the contrary of L is the contrary of the converse of L. Starting with [37], Charles Sanders Peirce created algebra from De Morgan's logic of relations, following the model of George Boole [7], [6], who created algebra from the logic of classes, "and after many attempts produced a good general algebra of logic, together with another algebra specially adapted to dyadic relations (*Studies in Logic*, by members of the Johns Hopkins University, 1883, Note B, 187–203). Schröder developed the last in a systematic manner" in [42] (quotation from [32]). F. W. K. Ernst Schröder's investigation of the calculus laid out by Peirce [39] in 17 pages extended to 649. His book remains today the only exhaustive treatise on the calculus of relations. For additional survey and historical material on relation algebras see [8], [12], [16], [17], [18], [26], [27], [28], [29], [30], [44], and [46].

Consider an arbitrary set, called the "universe of discourse" or simply the "universe". The universe could, depending on the situation and purposes, contain all possible mathematical objects, or all *states* of a machine, or all real numbers, or just a finite set of letters. The fundamental operations of the calculus of relations are natural set-theoretical operations on binary relations over the universe. In addition to the Boolean operations of union, intersection, and complementation, there are the "relative" (as Peirce calls them), or "Peircean" (as Tarski calls them) operations, namely the binary operation of "relative addition" (Peirce's name), the binary operation of "relative multiplication" (Peirce's name) or "composition" (De Morgan's name) and the unary operation of conversion. There are also four distinguished relations, namely

Key words and phrases. relation algebras, interval algebras, compass algebras, constraint satisfaction problems, relational semantics, predicate transformers, weakest preconditions.

the universal relation, the empty relation, the identity relation, and the diversity relation. The definitions of these operations and distinguished relations are listed below. In these definitions, x and y are arbitrary binary relations on the universe. By a ***binary relation*** we simply mean a set of ordered pairs. The ordered pair whose first element is p and whose second element is q is denoted $\langle p, q\rangle$.

union of x and y	$x + y = \{\langle p,q\rangle : \langle p,q\rangle \in x$ or $\langle p,q\rangle \in y\}$
intersection of x and y	$x \cdot y = \{\langle p,q\rangle : \langle p,q\rangle \in x$ and $\langle p,q\rangle \in y\}$
complement of x	$\overline{x} = \{\langle p,q\rangle : p, q$ are in the universe, but $\langle p,q\rangle \notin x\}$
relative sum of x and y	$x \dagger y = \{\langle p,r\rangle :$ for every q in the universe, $\langle p,q\rangle \in x$ or $\langle q,r\rangle \in y\}$
relative product of x and y	$x;y = \{\langle p,r\rangle :$ for some q, $\langle p,q\rangle \in x$ and $\langle q,r\rangle \in y\}$
converse of x	$\breve{x} = \{\langle q,p\rangle : \langle p,q\rangle \in x\}$
universal relation	$1 = \{\langle p,q\rangle : p, q$ are in the universe$\}$
empty relation	$0 = \emptyset$
identity relation	$1' = \{\langle p,p\rangle : p$ is in the universe$\}$
diversity relation	$0' = \{\langle p,q\rangle : p, q$ are in the universe, $p \neq q\}$

We are using nineteenth century notations. Both De Morgan and Peirce denoted the composition of x and y simply by "xy", but Schröder [42] used "$x;y$", as is done here. The notation "$x|y$" was used by Whitehead and Russell [53] and adopted by Tarski and his school [11]. Peirce introduced the notation "$\breve{x}$" for the converse of x. Schröder introduced "1'" and "0'" for the identity and diversity relations. Here are some laws in the calculus of relations. These laws hold for every possible universe, and all possible binary relations x, y, and z.

(i) $(x + y) + z = x + (y + z)$
(ii) $x + y = y + x$
(iii) $x = \overline{\overline{x} + y} + \overline{\overline{x} + \overline{y}}$
(iv) $x \cdot y = \overline{\overline{x} + \overline{y}}$
(v) $1 = x + \overline{x}$
(vi) $0 = \overline{1}$
(vii) $x;(y;z) = (x;y);z$
(viii) $x;1' = x$
(ix) $(x + y);z = x;z + y;z$
(x) $\breve{\breve{x}} = x$
(xi) $(x + y)\breve{} = \breve{x} + \breve{y}$
(xii) $(x;y)\breve{} = \breve{y};\breve{x}$
(xiii) $\breve{x};\overline{x;y} + \overline{y} = \overline{y}$
(xiv) $0' = \overline{1'}$
(xv) $x \dagger y = \overline{\overline{x};\overline{y}}$

A ***relation algebra*** is an algebra of the form

$$\mathfrak{A} = \langle A, +, \cdot, ^-, 0, 1, \dagger, ;, \breve{}, 0', 1'\rangle,$$

which satisfies the identities (i)-(xv) listed above. The first six identities say that $\langle A, +, \cdot, ^-, 0, 1\rangle$ is a Boolean algebra (called the ***Boolean part*** or ***Boolean reduct*** of $\mathfrak{A}$). One of the most significant laws of the calculus of relations is De Morgan's "Theorem K" (see [36, pp. 186–7, 224] or [30, p. 434–5]), which asserts that the following statements are equivalent:

$$x;y \leq z \qquad \breve{x};\overline{z} \leq \overline{y} \qquad \overline{z};\breve{y} \leq \overline{x}$$

After minor Boolean transformations Theorem K becomes the *cycle law*, that the following statements are equivalent:

$$x;y\cdot z=0 \qquad \breve{x};z\cdot y=0 \qquad z;\breve{y}\cdot x=0$$

The cycle law and De Morgan's Theorem K hold in every relation algebra because they can be proved from axioms (i)–(xv). There are many other equivalent axiomatizations for relation algebras. For example, equations (ix)–(xiii) can be replaced with the cycle law or with Theorem K.

The algebra containing all binary relations on the universe U is denoted $\mathfrak{Re}(U)$. Identities (i)–(xv) hold in $\mathfrak{Re}(U)$, so $\mathfrak{Re}(U)$ is a relation algebra. Relation algebras are defined by equations, so it follows that subalgebras, homomorphic images, and direct products of relation algebras are again relation algebras. The algebras that can be obtained from algebras of the form $\mathfrak{Re}(U)$ by forming subalgebras, homomorphic images, and direct products are called *representable* relation algebras. Roger Lyndon [22] showed that not all relation algebras are representable. It follows that the axioms (i)–(xv) are incomplete, in the sense that there are equations which hold in every algebra of the form $\mathfrak{Re}(U)$ but cannot be derived from (i)–(xv). J. Donald Monk [33] proved that the equations which hold in every algebra of the form $\mathfrak{Re}(U)$ cannot be derived from any finite set of equations.

For a relation algebra $\mathfrak{A}$, let $At\mathfrak{A}$ be the set of atoms of (the Boolean reduct of) $\mathfrak{A}$. (An element x of $\mathfrak{A}$ is an *atom* if $x\neq 0$ and, for every y in $\mathfrak{A}$, either $x\cdot y=x$ or $x\cdot y=0$.) If x is an atom of $\mathfrak{A}$, then so is $\breve{x}$. The relation algebra $\mathfrak{A}$ is said to be *atomic* if its Boolean reduct is atomic, that is, for every element y of $\mathfrak{A}$, if $y\neq 0$ then there is some atom x of $\mathfrak{A}$ such that $x\leq y$. Similarly, $\mathfrak{A}$ is said to be *complete* if its Boolean part is complete, that is, every subset X of $\mathfrak{A}$ has a least upper bound $\sum X$ and greatest lower bound $\prod X$. It turns out that if $\mathfrak{A}$ is both complete and atomic, then the structure of $\mathfrak{A}$ is entirely determined by its atoms and the action of the relative operations on the atoms. For a precise statement of this fact, define the *atom structure* of $\mathfrak{A}$ to be $\mathfrak{At}\mathfrak{A}=\langle At\mathfrak{A},C,\breve{\ },I\rangle$, where

$$C=\{\langle a,b,c\rangle : a,b,c\in At\mathfrak{A}\text{ and }a;b\geq c\} \quad\text{and}\quad I=\{a : a\in At\mathfrak{A}\text{ and }a\leq 1'\}.$$

For any atoms a,b,c of $\mathfrak{A}$, let

$$[a,b,c]=\{\,\langle a,b,c\rangle,\langle\breve{a},c,b\rangle,\langle b,\breve{c},\breve{a}\rangle,\langle\breve{b},\breve{a},\breve{c}\rangle,\langle\breve{c},a,\breve{b}\rangle,\langle c,\breve{b},a\rangle\,\}.$$

By the cycle law, C is a union of sets of the form $[a,b,c]$. We refer to such sets as *cycles*. Then the identity element, the converse of x, and the relative product of x and y can be computed from the atom structure according to

$$1'=\sum I \qquad \breve{x}=\sum\{\breve{a} : x\geq a\in At\mathfrak{A}\}$$
$$x;y=\sum\{c : \text{for some } a,b\in At\mathfrak{A},\ x\geq a,\ y\geq b,\ \langle a,b,c\rangle\in C\}$$

Hence to specify a complete atomic relation algebra it suffices to list its atoms, to list those atoms which are in I, to indicate which atoms are converses of which other atoms, and, finally, to list the cycles $[a,b,c]$. This is especially convenient when $\mathfrak{A}$ is finite. We present several examples of relation algebras using this method.

2. Interval algebras.

To define the interval algebra IA [1], [2], take the universe U to be the set of all "events", where an event is simply a pair of real numbers, the second of which is larger than the first. The first number in an event is its "starting time", the second its "ending time". (Our model for time here is just the real numbers.) Seven binary relations on events are defined in the list below, where x, x', y, y' are real numbers and $\langle x, x'\rangle$, $\langle y, y'\rangle$ are events.

$$
\begin{array}{ll}
\text{identity:} & 1' = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : x = y < x' = y'\} \\
\text{precedes:} & p = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : x < x' < y < y'\} \\
\text{during:} & d = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : y < x < x' < y'\} \\
\text{overlaps:} & o = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : x < y < x' < y'\} \\
\text{meets:} & m = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : x < x' = y < y'\} \\
\text{starts:} & s = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : x = y < x' < y'\} \\
\text{finishes:} & f = \{\langle\langle x,x'\rangle, \langle y,y'\rangle\rangle : y < x < x' = y'\}
\end{array}
$$

The seven relations listed above are studied in [50] and are used in some computer programs [5], [31], [43]. These relations generate a finite subalgebra of $\mathfrak{Re}(U)$, called the *interval algebra*, or simply the IA. The IA has 13 atoms, namely 1', p, $\breve{p}$, d, $\breve{d}$, o, $\breve{o}$, m, $\breve{m}$, s, $\breve{s}$, f, and $\breve{f}$. (It turns out that p alone will generate the IA, and so will each of the elements $\breve{p}$, m, $\breve{m}$, o, and $\breve{o}$ [21], [20, Theorem 4.4].) If we start with the rational numbers instead of the reals, or, in fact, any dense linear ordering without endpoints, then the resulting algebra is isomorphic to the IA. But if we use some other infinite linear ordering, then the relation algebra generated by 1', p, d, o, m, s, and f may not be finite, and the relations listed above may no longer be atoms. This happens, for example, when we use the integers. If we start with a finite linear ordering on U, then the subalgebra generated by 1', p, d, o, m, s, and f will be $\mathfrak{Re}(U)$. Any relation algebra obtained in this way will be called en interval algebra (while the IA is the one obtained from the reals or rationals). The IA has 75 cycles: $[1', 1', 1']$, $[1', s, s]$, $[1', m, m]$, $[1', p, p]$, $[1', o, o]$, $[1', f, f]$, $[1', d, d]$, $[s, 1', s]$, $[s, s, s]$, $[s, m, p]$, $[s, p, p]$, $[s, o, m]$, $[s, o, p]$, $[s, o, o]$, $[s, f, d]$, $[s, d, d]$, $[m, 1', m]$, $[m, s, m]$, $[m, m, p]$, $[m, p, p]$, $[m, o, p]$, $[m, f, s]$, $[m, f, o]$, $[m, f, d]$, $[m, d, s]$, $[m, d, o]$, $[m, d, d]$, $[p, 1', p]$, $[p, s, p]$, $[p, m, p]$, $[p, p, p]$, $[p, o, p]$, $[p, f, s]$, $[p, f, m]$, $[p, f, p]$, $[p, f, o]$, $[p, f, d]$, $[p, d, s]$, $[p, d, m]$, $[p, d, p]$, $[p, d, o]$, $[p, d, d]$, $[o, 1', o]$, $[o, s, o]$, $[o, m, p]$, $[o, p, p]$, $[o, o, m]$, $[o, o, p]$, $[o, o, o]$, $[o, f, s]$, $[o, f, o]$, $[o, f, d]$, $[o, d, s]$, $[o, d, o]$, $[o, d, d]$, $[f, 1', f]$, $[f, s, d]$, $[f, m, m]$, $[f, p, p]$, $[f, o, s]$ $[f, o, o]$, $[f, o, d]$, $[f, f, f]$, $[f, d, d]$, $[d, 1', d]$, $[d, s, d]$, $[d, m, p]$, $[d, p, p]$, $[d, o, s]$, $[d, o, m]$, $[d, o, p]$, $[d, o, o]$, $[d, o, d]$, $[d, f, d]$, $[d, d, d]$. Although all relative products in the IA can be computed from the cycles, it is convenient to also have the products listed in a table. The table of relative products of atoms of the IA is given in two parts (see Figs. 1 and 2). To save space the + signs are omitted, so, for example, $pdoms = p + d + o + m + s$. The table appeared first in [2]. It not only shows relative products of atoms in the IA, but also shows containments for the Allen-Hayes algebra [3], [4]. By the *Allen-Hayes algebra* we mean the direct product of "all" interval algebras, *i.e.*, the direct product of an indexed system of algebras containing one algebra from each isomorphism type of interval algebras. The Allen-Hayes algebra contains the elements 1', p, $\breve{p}$, d, $\breve{d}$, o, $\breve{o}$, m, $\breve{m}$, s, $\breve{s}$, f, and $\breve{f}$. They form a partition, *i.e.*, they are pairwise disjoint and $1 = p + \breve{p} + d + \breve{d} + o + \breve{o} + m + \breve{m} + s + \breve{s} + f + \breve{f}$. Finally, the relative product of any two of them is contained in (and not necessarily equal to) the corresponding entry in the table.

3. Compass algebras

Let the universe be the set of all points in the n-dimensional Euclidean space $\mathbb{R}^n$, where $\mathbb{R}$ is the set of real numbers. Let $\mathbb{R}^+$ be the set of positive real numbers. For every vector $\mathbf{v}$ in $\mathbb{R}^n$ define two binary relations on $\mathbb{R}^n$ as follows:

$$D_{\mathbf{v}} = \{\langle \mathbf{x}, \mathbf{y}\rangle : \mathbf{x}, \mathbf{y} \in \mathbb{R}^n \text{ and for some } r \text{ in } \mathbb{R}^+, \mathbf{x} + r\mathbf{v} = \mathbf{y}\},$$

$$E_{\mathbf{v}} = \{\langle \mathbf{x}, \mathbf{y}\rangle : \mathbf{x}, \mathbf{y} \in \mathbb{R}^n \text{ and for some } r \text{ in } \mathbb{R}, \mathbf{x} + r\mathbf{v} = \mathbf{y}\}.$$

Here are some easily proved properties of these relations.

Theorem 1. (i) $D_{\mathbf{0}} = E_{\mathbf{0}} = 1' = \{\langle \mathbf{x}, \mathbf{x}\rangle : \mathbf{x} \in \mathbb{R}^n\}$,

	$1'$	p	$\breve{p}$	d	$\breve{d}$	o	$\breve{o}$
$1'$	$1'$	p	$\breve{p}$	d	$\breve{d}$	o	$\breve{o}$
p	p	p	1	$pdoms$	p	p	$pdoms$
$\breve{p}$	$\breve{p}$	1	$\breve{p}$	$\breve{p}d\breve{o}\breve{m}f$	$\breve{p}$	$\breve{p}d\breve{o}\breve{m}f$	$\breve{p}$
d	d	p	$\breve{p}$	d	1	$pdoms$	$\breve{p}d\breve{o}\breve{m}f$
$\breve{d}$	$\breve{d}$	$p\breve{d}om\breve{f}$	$\breve{p}\breve{d}\breve{o}\breve{m}\breve{s}$	$1'd\breve{d}o\breve{o}s\breve{s}f\breve{f}$	$\breve{d}$	$\breve{d}o\breve{f}$	$\breve{d}\breve{o}\breve{s}$
o	o	p	$\breve{p}\breve{d}\breve{o}\breve{m}\breve{s}$	dos	$p\breve{d}om\breve{f}$	pom	$1'd\breve{d}o\breve{o}s\breve{s}f\breve{f}$
$\breve{o}$	$\breve{o}$	$p\breve{d}om\breve{f}$	$\breve{p}$	$d\breve{o}f$	$\breve{p}\breve{d}\breve{o}\breve{m}\breve{s}$	$1'd\breve{d}o\breve{o}s\breve{s}f\breve{f}$	$\breve{p}\breve{o}\breve{m}$
m	m	p	$\breve{p}\breve{d}\breve{o}\breve{m}\breve{s}$	dos	p	p	dos
$\breve{m}$	$\breve{m}$	$p\breve{d}om\breve{f}$	$\breve{p}$	$d\breve{o}f$	$\breve{p}$	$d\breve{o}f$	$\breve{p}$
s	s	p	$\breve{p}$	d	$p\breve{d}om\breve{f}$	pom	$d\breve{o}f$
$\breve{s}$	$\breve{s}$	$p\breve{d}om\breve{f}$	$\breve{p}$	$d\breve{o}f$	$\breve{d}$	$\breve{d}o\breve{f}$	$\breve{o}$
f	f	p	$\breve{p}$	d	$\breve{p}\breve{d}\breve{o}\breve{m}\breve{s}$	dos	$\breve{p}\breve{o}\breve{m}$
$\breve{f}$	$\breve{f}$	p	$\breve{p}\breve{d}\breve{o}\breve{m}\breve{s}$	dos	$\breve{d}$	o	$\breve{d}\breve{o}\breve{s}$

FIGURE 1. The interval algebra products, first part.

	m	$\breve{m}$	s	$\breve{s}$	f	$\breve{f}$
$1'$	m	$\breve{m}$	s	$\breve{s}$	f	$\breve{f}$
p	p	$pdoms$	p	p	$pdoms$	p
$\breve{p}$	$\breve{p}d\breve{o}\breve{m}f$	$\breve{p}$	$\breve{p}d\breve{o}\breve{m}f$	$\breve{p}$	$\breve{p}$	$\breve{p}$
d	p	$\breve{p}$	d	$\breve{p}d\breve{o}\breve{m}f$	d	$pdoms$
$\breve{d}$	$\breve{d}o\breve{f}$	$\breve{d}\breve{o}\breve{s}$	$\breve{d}o\breve{f}$	$\breve{d}$	$\breve{d}\breve{o}\breve{s}$	$\breve{d}$
o	p	$\breve{d}\breve{o}\breve{s}$	o	$\breve{d}o\breve{f}$	dos	pom
$\breve{o}$	$\breve{d}o\breve{f}$	$\breve{p}$	$d\breve{o}f$	$\breve{p}\breve{o}\breve{m}$	$\breve{o}$	$\breve{d}\breve{o}\breve{s}$
m	p	$1'f\breve{f}$	m	m	dos	p
$\breve{m}$	$1's\breve{s}$	$\breve{p}$	$d\breve{o}f$	$\breve{p}$	$\breve{m}$	$\breve{m}$
s	p	$\breve{m}$	s	$1's\breve{s}$	d	pom
$\breve{s}$	$\breve{d}o\breve{f}$	$\breve{m}$	$1's\breve{s}$	$\breve{s}$	$\breve{o}$	$\breve{d}$
f	m	$\breve{p}$	d	$\breve{p}\breve{o}\breve{m}$	f	$1'f\breve{f}$
$\breve{f}$	m	$\breve{d}\breve{o}\breve{s}$	o	$\breve{d}$	$1'f\breve{f}$	$\breve{f}$

FIGURE 2. The interval algebra products, second part.

(ii) $\breve{D}_{\mathbf{v}} = D_{-\mathbf{v}} = \{\, \langle \mathbf{x},\mathbf{y} \rangle : \textit{for some } r \textit{ in } \mathbb{R}^+,\ \mathbf{x} - r\mathbf{v} = \mathbf{y} \,\}$,
(iii) $D_{\mathbf{v}} ; D_{\mathbf{v}} = D_{\mathbf{v}}$,
(iv) $D_{\mathbf{v}} = D_{r\mathbf{v}}$ and $E_{\mathbf{v}} = E_{r\mathbf{v}}$ whenever $r \in \mathbb{R}^+$,
(v) $E_{\mathbf{v}} = \breve{E}_{\mathbf{v}} = E_{\mathbf{v}} ; E_{\mathbf{v}} = D_{\mathbf{v}} ; \breve{D}_{\mathbf{v}} = \breve{D}_{\mathbf{v}} , D_{\mathbf{v}} = D_{\mathbf{v}} + \breve{D}_{\mathbf{v}} + D_{\mathbf{0}}$,
(vi) $E_{\mathbf{v}}$ *is an equivalence relation on* $\mathbb{R}^n$,
(vii) $D_{\mathbf{v}} ; D_{\mathbf{w}} = D_{\mathbf{w}} ; D_{\mathbf{v}} = \{\langle \mathbf{x},\mathbf{y} \rangle : \textit{for some } r, s \textit{ in } \mathbb{R}^+,\ \mathbf{x} + r\mathbf{v} + s\mathbf{w} = \mathbf{y}\}$,
(viii) $E_{\mathbf{v}} ; E_{\mathbf{w}} = E_{\mathbf{w}} ; E_{\mathbf{v}} = \{\langle \mathbf{x},\mathbf{y} \rangle : \textit{for some } r, s \textit{ in } \mathbb{R}, \mathbf{x} + r\mathbf{v} + s\mathbf{w} = \mathbf{y}\}$,
(ix) $\langle \mathbf{x},\mathbf{y} \rangle \in E_{\mathbf{v}}$ *iff* $\mathbf{y} - \mathbf{x}$ *is in the subspace spanned by* $\mathbf{v}$,
(x) $\langle \mathbf{x},\mathbf{y} \rangle \in E_{\mathbf{v}} ; E_{\mathbf{w}}$ *iff* $\mathbf{y} - \mathbf{x}$ *is in the subspace spanned by* $\mathbf{v}$ *and* $\mathbf{w}$,
(xi) $\langle \mathbf{x},\mathbf{y} \rangle \in E_{\mathbf{v}_0} ; \ldots ; E_{\mathbf{v}_m}$ *iff* $\mathbf{y} - \mathbf{x}$ *is in the subspace spanned by* $\mathbf{v}_0, \ldots, \mathbf{v}_m$.

For any m vectors $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^n$, let $\mathfrak{C}_n[\mathbf{v}_1, \ldots, \mathbf{v}_m]$ be the subalgebra of $\mathfrak{Re}(\mathbb{R}^n)$ generated by the relations $D_{\mathbf{v}_1}, \ldots, D_{\mathbf{v}_m}$. $\mathfrak{C}_n[\mathbf{v}_1, \ldots, \mathbf{v}_m]$ is called the *n-dimensional compass algebra determined by* $\mathbf{v}_0, \ldots, \mathbf{v}_m$. If $\mathbf{v}$ and $\mathbf{w}$ are a linearly dependent pair of nonzero vectors, then either $D_{\mathbf{v}} = D_{\mathbf{w}}$ or $D_{\mathbf{v}} = \breve{D}_{\mathbf{w}}$. If $\mathbf{v}$ and $\mathbf{w}$ both appear in a list of vectors generating a compass algebra, then $\mathbf{v}$ can be deleted from the list, and the same compass algebra will still be obtained from the remaining vectors. Even if the vectors are pairwise linearly independent, deleting one of them may not result in a strictly smaller compass algebra. The structure of $\mathfrak{C}_n[\mathbf{v}_1, \ldots, \mathbf{v}_m]$ depends heavily on the choice of vectors. But if $\mathbf{v}_1, \ldots, \mathbf{v}_m$ is a linear independent set of vectors, then the structure of $\mathfrak{C}_n[\mathbf{v}_1, \ldots, \mathbf{v}_m]$ is completely determined by m. More exactly, if $\mathbf{v}_1, \ldots, \mathbf{v}_m$ and $\mathbf{v}'_1, \ldots, \mathbf{v}'_m$ are two linearly independent sets of vectors in $\mathbb{R}^n$ (hence $m \leq n$), then $\mathfrak{C}_n[\mathbf{v}_1, \ldots, \mathbf{v}_m]$ is isomorphic to $\mathfrak{C}_n[\mathbf{v}'_1, \ldots, \mathbf{v}'_m]$.

4. Examples of compass algebras

The 1-dimensional compass algebra $\mathfrak{C}_1[\langle 1 \rangle]$ generated by the 1-dimensional vector $\langle 1 \rangle$ has three atoms, namely $D_{\langle 1 \rangle}$, $D_{\langle -1 \rangle}$, and $D_{\langle 0 \rangle}$. $\mathfrak{C}_1[\langle 1 \rangle]$ is known as the "Point Algebra" [19], [21], [20], [47], [48], [49], [51], [52]. For a description of the structure of $\mathfrak{C}_1[\langle 1 \rangle]$ in terms of atoms and cycles, let $1' = D_{\langle 0 \rangle}$, $a = D_{\langle 1 \rangle}$, and $\breve{a} = D_{\langle -1 \rangle}$. Then the cycles of $\mathfrak{A}$ are $[1', 1', 1']$, $[1', a, a]$, $[a, 1', a]$, and $[a, a, a]$. The table of relative products of atoms is

	$1'$	a	$\breve{a}$
$1'$	$1'$	a	$\breve{a}$
a	a	a	1
$\breve{a}$	$\breve{a}$	1	a

Every 1-dimensional vector in 1-space must determine one of the relations $D_{\langle 1 \rangle}$, $D_{\langle -1 \rangle}$, or $D_{\langle 0 \rangle}$, so no new 1-dimensional compass algebras are obtained by considering two or more vectors in 1-dimensional space. However, there is one other 1-dimensional compass algebra, namely $\mathfrak{C}_1[\langle 0 \rangle]$. This algebra has two atoms, namely $D_{\langle 0 \rangle} = 1'$ and $D_{\langle 1 \rangle} + D_{\langle -1 \rangle} = 0'$. The cycles of $\mathfrak{A}$ are $[1', 1', 1']$, $[1', 0', 0']$, and $[0', 0', 0']$, and the table of relative products of atoms is

	$1'$	$0'$
$1'$	$1'$	$0'$
$0'$	$0'$	1

By comparing this and the previous table it can be seen that $\mathfrak{C}_1[\langle 0 \rangle]$ is isomorphic to a subalgebra of $\mathfrak{C}_1[\langle 1 \rangle]$, the one with atoms $1'$ and $a + \breve{a}$. Also, $\mathfrak{C}_1[\langle 0 \rangle]$ is isomorphic to $\mathfrak{C}_n[\langle 0 \rangle]$ for every integer n.

Now we consider 2-dimensional compass algebras. Among these are particular algebras which inspired the name "compass algebra". We start with the compass algebra $\mathfrak{C}_2[\langle 1, 0 \rangle, \langle 0, 1 \rangle]$. We would get the same algebra with any two linearly independent vectors in $\mathbb{R}^2$, but these two allow us to dub $E_{\langle 1,0 \rangle}$ the "east-west" direction, while $E_{\langle 0,1 \rangle}$ is the "north-south" direction. Thus $C_2[\langle 0, 1 \rangle, \langle 1, 0 \rangle]$ is a "2-directional" algebra of relations. "East", "west", "north", and "south"

	$1'$	a	$\breve{a}$	b	$\breve{b}$	c	$\breve{c}$	d	$\breve{d}$
$1'$	$1'$	a	$\breve{a}$	b	$\breve{b}$	c	$\breve{c}$	d	$\breve{d}$
a	a	a	$1'a\breve{a}$	b	$\breve{b}\breve{c}\breve{d}$	b	$\breve{d}$	bcd	$\breve{d}$
b	b	b	bcd	b	1	b	$ab\breve{d}$	bcd	$ab\breve{d}$
c	c	b	d	b	$d\breve{a}\breve{b}$	c	$1'c\breve{c}$	d	$ab\breve{d}$
d	d	bcd	d	bcd	$d\breve{a}\breve{b}$	d	$d\breve{a}\breve{b}$	d	1
$\breve{a}$	$\breve{a}$	$1'a\breve{a}$	$\breve{a}$	bcd	$\breve{b}$	d	$\breve{b}$	d	$\breve{b}\breve{c}\breve{d}$
$\breve{b}$	$\breve{b}$	$\breve{b}\breve{c}\breve{d}$	$\breve{b}$	1	$\breve{b}$	$d\breve{a}\breve{b}$	$\breve{b}$	$d\breve{a}\breve{b}$	$\breve{b}\breve{c}\breve{d}$
$\breve{c}$	$\breve{c}$	$\breve{d}$	$\breve{b}$	$ab\breve{d}$	$\breve{b}$	$1'c\breve{c}$	$\breve{c}$	$d\breve{a}\breve{b}$	$\breve{d}$
$\breve{d}$	$\breve{d}$	$\breve{d}$	$\breve{b}\breve{c}\breve{d}$	$ab\breve{d}$	$\breve{b}\breve{c}\breve{d}$	$ab\breve{d}$	$\breve{d}$	1	$\breve{d}$

FIGURE 3. Products for $\mathfrak{C}_2[\langle 1,0\rangle, \langle 0,1\rangle]$

are the relations $D_{\langle 1,0\rangle}$, $D_{\langle -1,0\rangle}$, $D_{\langle 0,1\rangle}$, and $D_{\langle 0,-1\rangle}$, respectively. In the standard Euclidean plane of analytic geometry, the points "east" of the origin are all the points on the positive part of the x-axis, and so on. $\mathfrak{C}_2[\langle 1,0\rangle, \langle 0,1\rangle]$ has nine atoms, namely $D_{\langle 0,0\rangle}$, $D_{\langle 1,0\rangle}$, $D_{\langle -1,0\rangle}$, $D_{\langle 0,1\rangle}$, $D_{\langle 0,-1\rangle}$, $D_{\langle 0,1\rangle};D_{\langle 1,0\rangle}$, $D_{\langle 0,1\rangle};D_{\langle -1,0\rangle}$, $D_{\langle 0,-1\rangle};D_{\langle 1,0\rangle}$, and $D_{\langle 0,-1\rangle};D_{\langle 0,-1\rangle}$. The last four atoms could be called "northeasterly", "northwesterly", "southeasterly", and "southwesterly", respectively, since they do not correspond exactly with directions of the compass. The points in the Euclidean plane which can be reached by going northeasterly from the origin are exactly those in the first quadrant. Let

$1' = D_{\langle 0,0\rangle}$ = identity	
$a = D_{\langle 1,0\rangle}$ = east	$b = a;c = c;a$ = northeasterly
$\breve{a} = D_{\langle -1,0\rangle}$ = west	$\breve{b} = \breve{c};\breve{a} = \breve{a};\breve{c}$ = southwesterly
$c = D_{\langle 0,1\rangle}$ = north	$d = \breve{a};c = c;\breve{a}$ = northwesterly
$\breve{c} = D_{\langle 0,-1\rangle}$ = south	$\breve{d} = \breve{c};a = a;\breve{c}$ = southeasterly

Then the 33 cycles of $\mathfrak{A}$ are $[1',1',1']$, $[1',a,a]$, $[a,1',a]$, $[1',b,b]$, $[b,1',b]$, $[1',c,c]$, $[c,1',c]$, $[1',d,d]$, $[d,1',d]$, $[a,a,a]$, $[a,b,b]$, $[a,c,b]$, $[a,d,b]$, $[a,d,c]$, $[a,d,d]$, $[b,a,b]$, $[b,b,b]$, $[b,c,b]$, $[b,d,b]$, $[b,d,c]$, $[b,d,d]$, $[c,a,b]$, $[c,b,b]$, $[c,c,c]$, $[c,d,d]$, $[d,a,b]$, $[d,a,c]$, $[d,a,d]$, $[d,b,b]$, $[d,b,c]$, $[d,b,d]$, $[d,c,d]$, $[d,d,d]$. The relative products of atoms are given in Fig. 3.

The compass algebra $\mathfrak{C}_2[\langle 1,0\rangle, \langle 1,1\rangle, \langle 0,1\rangle]$ has 13 atoms, namely $1'$, a, b, c, d, e, f, $\breve{a}$, $\breve{b}$, $\breve{c}$, $\breve{d}$, $\breve{e}$, and $\breve{f}$, where $1' = D_{\langle 0,0\rangle}$, $a = D_{\langle 1,0\rangle}$, $b = D_{\langle 1,0\rangle};D_{\langle 1,1\rangle}$, $c = D_{\langle 1,1\rangle}$, $d = D_{\langle 1,1\rangle};D_{\langle 0,1\rangle}$, $e = D_{\langle 0,1\rangle}$, and $f = D_{\langle 0,1\rangle};D_{\langle -1,0\rangle}$. There are 89 cycles, each having the form $[x,y,z]$ with x,y,z in $\{1',a,b,c,d,e,f\}$. The cycles are not listed, but they can be read from the table of relative products in Fig. 4. Set $\hat{x} = x + \breve{x}$ for every x in $\mathfrak{C}_2[\langle 1,0\rangle, \langle 1,1\rangle, \langle 0,1\rangle]$. Then $1' + \hat{a} = E_{\langle 1,0\rangle}$, $1' + \hat{c} = E_{\langle 1,1\rangle}$, $1' + \hat{e} = E_{\langle 0,1\rangle}$, *etc.* and $1'$, $\hat{a}$, $\hat{b}$, $\hat{c}$, $\hat{d}$, $\hat{e}$, and $\hat{f}$ are the atoms of a subalgebra called the "symmetric subalgebra" of $\mathfrak{C}_2[\langle 1,0\rangle, \langle 1,1\rangle, \langle 0,1\rangle]$. The table of products for this subalgebra is

	$1'$	$\hat{a}$	$\hat{b}$	$\hat{c}$	$\hat{d}$	$\hat{e}$	$\hat{f}$
$1'$	$1'$	$\hat{a}$	$\hat{b}$	$\hat{c}$	$\hat{d}$	$\hat{e}$	$\hat{f}$
$\hat{a}$	$\hat{a}$	$1'\hat{a}$	$\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{b}\hat{d}\hat{e}\hat{f}$	$\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{b}\hat{c}\hat{d}\hat{f}$	$\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$
$\hat{b}$	$\hat{b}$	$\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	1	$\hat{a}\hat{b}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$
$\hat{c}$	$\hat{c}$	$\hat{b}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{d}\hat{e}\hat{f}$	$1'\hat{c}$	$\hat{a}\hat{b}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{d}\hat{f}$	$\hat{a}\hat{b}\hat{d}\hat{e}\hat{f}$
$\hat{d}$	$\hat{d}$	$\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{d}\hat{e}\hat{f}$	1	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$
$\hat{e}$	$\hat{e}$	$\hat{b}\hat{c}\hat{d}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{f}$	$\hat{a}\hat{b}\hat{d}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{f}$	$1'\hat{e}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{f}$
$\hat{f}$	$\hat{f}$	$\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{e}\hat{f}$	$\hat{a}\hat{b}\hat{c}\hat{d}\hat{f}$	1

	$1'$	a	$\breve{a}$	b	$\breve{b}$	c	$\breve{c}$	d	$\breve{d}$	e	$\breve{e}$	f	$\breve{f}$
$1'$	$1'$	a	$\breve{a}$	b	$\breve{b}$	c	$\breve{c}$	d	$\breve{d}$	e	$\breve{e}$	f	$\breve{f}$
a	a	a	$1'a\breve{a}$	b	$\breve{b}\breve{c}\breve{d}\breve{e}\breve{f}$	b	$\breve{d}\breve{e}\breve{f}$	bcd	$\breve{d}\breve{e}\breve{f}$	bcd	$\breve{f}$	$bcdef$	$\breve{f}$
$\breve{a}$	$\breve{a}$	$1'a\breve{a}$	$\breve{a}$	$bcdef$	$\breve{b}$	def	$\breve{b}$	def	$\breve{b}\breve{c}\breve{d}$	f	$\breve{b}\breve{c}\breve{d}$	f	$\breve{b}\breve{c}\breve{d}\breve{e}\breve{f}$
b	b	b	$bcdef$	b	1	b	$ab\breve{d}\breve{e}\breve{f}$	bcd	$ab\breve{d}\breve{e}\breve{f}$	bcd	$ab\breve{f}$	$bcdef$	$ab\breve{f}$
$\breve{b}$	$\breve{b}$	$\breve{b}\breve{c}\breve{d}\breve{e}\breve{f}$	$\breve{b}$	1	$\breve{b}$	$def\breve{a}\breve{b}$	$\breve{b}$	$def\breve{a}\breve{b}$	$\breve{b}\breve{c}\breve{d}$	$f\breve{a}\breve{b}$	$\breve{b}\breve{c}\breve{d}$	$f\breve{a}\breve{b}$	$\breve{b}\breve{c}\breve{d}\breve{e}\breve{f}$
c	c	b	def	b	$def\breve{a}\breve{b}$	c	$1'c\breve{c}$	d	$abd\breve{e}f$	d	$ab\breve{f}$	def	$ab\breve{f}$
$\breve{c}$	$\breve{c}$	$\breve{d}\breve{e}\breve{f}$	$\breve{b}$	$ab\breve{d}\breve{e}\breve{f}$	$\breve{b}$	$1'c\breve{c}$	$\breve{c}$	$def\breve{a}\breve{b}$	$\breve{d}$	$f\breve{a}\breve{b}$	$\breve{d}$	$f\breve{a}\breve{b}$	$\breve{d}\breve{e}\breve{f}$
d	d	bcd	def	bcd	$def\breve{a}\breve{b}$	d	$def\breve{a}\breve{b}$	d	1	d	$abcd\breve{f}$	def	$abcd\breve{f}$
$\breve{d}$	$\breve{d}$	$\breve{d}\breve{e}\breve{f}$	$\breve{b}\breve{c}\breve{d}$	$ab\breve{d}\breve{e}\breve{f}$	$\breve{b}\breve{c}\breve{d}$	$ab\breve{d}\breve{e}f$	$\breve{d}$	1	$\breve{d}$	$f\breve{a}\breve{b}\breve{c}\breve{d}$	$\breve{d}$	$f\breve{a}\breve{b}\breve{c}\breve{d}$	$\breve{d}\breve{e}\breve{f}$
e	e	bcd	f	bcd	$f\breve{a}\breve{b}$	d	$f\breve{a}\breve{b}$	d	$f\breve{a}\breve{b}\breve{c}\breve{d}$	e	$1'e\breve{e}$	f	$abcd\breve{f}$
$\breve{e}$	$\breve{e}$	$\breve{f}$	$\breve{b}\breve{c}\breve{d}$	$ab\breve{f}$	$\breve{b}\breve{c}\breve{d}$	$ab\breve{f}$	$\breve{d}$	$abcd\breve{f}$	$\breve{d}$	$1'e\breve{e}$	$\breve{e}$	$f\breve{a}\breve{b}\breve{c}\breve{d}$	$\breve{f}$
f	f	$bcdef$	f	$bcdef$	$f\breve{a}\breve{b}$	def	$f\breve{a}\breve{b}$	def	$f\breve{a}\breve{b}\breve{c}\breve{d}$	f	$f\breve{a}\breve{b}\breve{c}\breve{d}$	f	1
$\breve{f}$	$\breve{f}$	$\breve{f}$	$\breve{b}\breve{c}\breve{d}\breve{e}\breve{f}$	$ab\breve{f}$	$\breve{b}\breve{c}\breve{d}\breve{e}\breve{f}$	$ab\breve{f}$	$\breve{d}\breve{e}\breve{f}$	$abcd\breve{f}$	$\breve{d}\breve{e}\breve{f}$	$abcd\breve{f}$	$\breve{f}$	1	$\breve{f}$

FIGURE 4. Products for $\mathfrak{C}_2[\langle 1,0\rangle, \langle 1,1\rangle, \langle 0,1\rangle]$

Next we consider the 2-dimensional compass algebra $\mathfrak{C}_2[\langle 1,0\rangle, \langle 1,1\rangle, \langle 0,1\rangle, \langle -1,1\rangle]$. Besides the directions "east-west" $E_{\langle 1,0\rangle}$ and "north-south" $E_{\langle 0,1\rangle}$, this algebra has directions "northeast-southwest" $E_{\langle 1,1\rangle}$ and "southeast-northwest" $E_{\langle -1,1\rangle}$. There are 17 atoms, namely $D_{\langle 0,0\rangle}$ and 16 others, which are listed counterclockwise, starting at the x-axis: $D_{\langle 1,0\rangle}$, $D_{\langle 1,0\rangle} ; D_{\langle 1,1\rangle}$, $D_{\langle 1,1\rangle}$, $D_{\langle 1,1\rangle} ; D_{\langle 0,1\rangle}$, $D_{\langle 0,1\rangle}$, $D_{\langle 0,1\rangle} ; D_{\langle -1,1\rangle}$, $D_{\langle -1,1\rangle}$, $D_{\langle -1,1\rangle} ; D_{\langle -1,0\rangle}$, $D_{\langle -1,0\rangle}$, $D_{\langle -1,0\rangle} ; D_{\langle -1,-1\rangle}$, $D_{\langle -1,-1\rangle}$, $D_{\langle -1,-1\rangle} ; D_{\langle 0,-1\rangle}$, $D_{\langle 0,-1\rangle}$, $D_{\langle 0,-1\rangle} ; D_{\langle 1,-1\rangle}$, $D_{\langle 1,-1\rangle}$, $D_{\langle 1,-1\rangle} ; D_{\langle 1,0\rangle}$.

The 2-dimensional compass algebra $\mathfrak{C}_2[\langle 1,0\rangle]$ has just four atoms, namely $D_{\langle 0,0\rangle}$, $D_{\langle 1,0\rangle}$, $D_{\langle -1,0\rangle}$, and $F = (\mathbb{R}^2 \times \mathbb{R}^2) \cdot \overline{D_{\langle 0,0\rangle} + D_{\langle 1,0\rangle} + D_{\langle -1,0\rangle}}$. Note that F is a symmetric relation, *i.e.*, $\breve{F} = F$, unlike $D_{\langle 1,0\rangle}$ or $D_{\langle -1,0\rangle}$. The points of the plane which are in the relation F to the origin are all those which lie in the upper half plane or lower half plane (*i.e.*, not on the x-axis). Let $1' = D_{\langle 0,0\rangle}$, $a = D_{\langle 1,0\rangle}$, $\breve{a} = D_{\langle -1,0\rangle}$, and $b = \overline{1' + a + \breve{a}}$. Then the cycles of $\mathfrak{C}_2[\langle 1,0\rangle]$ are $[1',1',1']$, $[1',a,a]$, $[a,1',a]$, $[1',b,b]$, $[a,a,a]$, $[a,b,b]$, $[b,b,b]$, and the relative products of atoms are:

	$1'$	a	$\breve{a}$	b
$1'$	$1'$	a	$\breve{a}$	b
a	a	a	$1'a\breve{a}$	b
$\breve{a}$	$\breve{a}$	$1'a\breve{a}$	$\breve{a}$	b
b	b	b	b	1

This algebra illustrates a general phenomenon. If $\mathbf{v}_1, \dots, \mathbf{v}_m \in \mathbb{R}^n$ are pairwise linearly independent but do not span $\mathbb{R}^n$, then $\mathfrak{C}_n[\mathbf{v}_1, \dots, \mathbf{v}_m]$ will have only one atom for the subspace orthogonal to the subspace spanned by $\mathbf{v}_1 \dots, \mathbf{v}_m$. Notice that this situation must arise whenever the number of directions is less than the number of dimensions, *i.e.*, whenever $m < n$.

Now we consider 3-dimensional compass algebras. Let $\mathbf{u} = \langle 1,0,0\rangle$, $\mathbf{v} = \langle 0,1,0\rangle$, $\mathbf{w} = \langle 1,1,0\rangle$, $\mathbf{x} = \langle -1,1,0\rangle$ and $\mathbf{y} = \langle 0,0,1\rangle$. The 3-dimensional compass algebra generated by a single vector in $\{\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x},\mathbf{y}\}$ has 4 atoms. The algebra generated by any two vectors in $\{\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x},\mathbf{y}\}$ has 10 atoms. Note that $\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x}$ all lie in the same 2-dimensional subspace. Hence any three vectors in $\{\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x}\}$ generate a 3-dimensional compass algebra with 14 atoms, while $\mathfrak{C}_3[\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x}]$ has 18 atoms. The vector $\mathbf{y}$ and any two vectors in $\{\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x}\}$ form a linearly independent set, and generate a compass algebra with 27 atoms. The vector $\mathbf{y}$ and any three vectors in $\{\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x}\}$ generate a compass algebra with 39 atoms. Finally, $\mathfrak{C}_3[\mathbf{u},\mathbf{v},\mathbf{w},\mathbf{x},\mathbf{y}]$ has 51 atoms.

Not every compass algebra determined by a finite set of vectors is finite. Let $\mathbf{z} = \langle 1,1,1\rangle$. Then $\mathfrak{C}_3[\mathbf{u},\mathbf{v},\mathbf{y},\mathbf{z}] = \mathfrak{C}_3[\langle 1,0,0\rangle, \langle 0,1,0\rangle, \langle 0,0,1\rangle, \langle 1,1,1\rangle]$ is infinite. To see this, let $X_0 = E_{\mathbf{u}}$,

$Y_0 = E_{\mathbf{v}}$, $Z_0 = E_{\mathbf{y}}$, and, for every integer n, $X_{n+1} = X_n;E_{\mathbf{z}} \cdot Y_n;Z_n$, $Y_{n+1} = Y_n;E_{\mathbf{z}} \cdot X_n;Z_n$, and $Z_{n+1} = Z_n;E_{\mathbf{z}} \cdot X_n;Y_n$. Then X_n, Y_n, and Z_n are all distinct equivalence relations for every n. In particular,

$$\begin{array}{lll} X_0 = E_{(1,0,0)} & Y_0 = E_{(0,1,0)} & Z_0 = E_{(0,0,1)} \\ X_1 = E_{(0,1,1)} & Y_1 = E_{(1,0,1)} & Z_1 = E_{(1,1,0)} \\ X_2 = E_{(2,1,1)} & Y_2 = E_{(1,2,1)} & Z_2 = E_{(1,1,2)} \\ X_3 = E_{(2,3,3)} & Y_3 = E_{(3,2,3)} & Z_3 = E_{(3,3,2)} \\ X_4 = E_{(6,5,5)} & Y_4 = E_{(5,6,5)} & Z_4 = E_{(5,5,6)} \\ \vdots & \vdots & \vdots \end{array}$$

5. Some NP-complete constraint satisfiability problems

Let $\mathfrak{A}$ be a relation algebra. An *$\mathfrak{A}$-matrix* is a matrix of elements of $\mathfrak{A}$. Suppose M is an n-by-n $\mathfrak{A}$-matrix. We say M is *zeroless* if no entry in M is 0, and M is *closed* if $M_{ii} \leq 1'$, $(M_{ij})^{\smile} = M_{ji}$, and $M_{ij};M_{jk} \leq M_{ik}$ whenever $1 \leq i,j,k \leq n$. If N is another n-by-n matrix, we say N is a *reduction* of M, in symbols, $N \leq M$, if $N_{ij} \leq M_{ij}$ whenever $1 \leq i,j \leq n$. If X is a set of elements of $\mathfrak{A}$, we say M is *bounded by* X if every entry of M is included in some element of X.

A *binary constraint matrix* is a matrix of binary relations. An n-by-n binary constraint matrix M determines an n-ary relation $R(M) = \{\langle p_1,\ldots,p_n\rangle : \langle p_i,p_j\rangle \in M_{ij}$ whenever $1 \leq i,j \leq n\}$. The matrix M specifies a *binary constraint problem.* The *solutions* to this problem are the n-tuples in $R(M)$, and the problem is *solvable* if it has a solution. Let U be the set of elements that appear in any pair in any relation in M. Each n-tuple $\langle p_1,\ldots,p_n\rangle$ of elements of U corresponds naturally to an n-by-n matrix N of atoms of $\mathfrak{Re}(U)$, where $N_{ij} = \{\langle p_i,p_j\rangle\}$ whenever $1 \leq i,j \leq n$. Note that $\langle p_1,\ldots,p_n\rangle$ is a solution to M if and only if its corresponding matrix N is a reduction of M. Furthermore, as a binary constraint problem, M has a solution just in case there is a closed zeroless reduction of M bounded by the set of atoms of $\mathfrak{Re}(U)$.

This last observation permits us to generalize the concept of constraint satisfaction to arbitrary atomic relation algebras. Let $\mathfrak{A}$ be an atomic relation algebra and let M be an $\mathfrak{A}$-matrix. We say that M is *proto-solvable over* $\mathfrak{A}$ if there is a closed zeroless reduction of M which is bounded by the set of atoms of $\mathfrak{A}$. Note that if N is a closed zeroless $\mathfrak{A}$-matrix bounded by the atoms of $\mathfrak{A}$, then all the entries in N must actually be atoms of $\mathfrak{A}$. Such a matrix, whose entries are all atoms of $\mathfrak{A}$, is called *atomic.* So the $\mathfrak{A}$-matrix M is proto-solvable if it has a closed atomic reduction N. Such an N is called a *proto-solution.* The *constraint satisfiability problem* for an atomic relation algebra $\mathfrak{A}$ is this: given an $\mathfrak{A}$-matrix, determine whether it has a proto-solution.

For an $\mathfrak{Re}(U)$-matrix M, the solutions and proto-solutions (over $\mathfrak{Re}(U)$) are in a one-to-one correspondence, as observed above. But for matrices over atomic subalgebras of $\mathfrak{Re}(U)$, such a correspondence may not exist. Indeed, it is easy to find a set U, a finite subalgebra $\mathfrak{A}$ of $\mathfrak{Re}(U)$, and an $\mathfrak{A}$-matrix M such that M has a proto-solution but no solution. For example, let $U = \{1,2,3\}$, let $\mathfrak{A}$ be the subalgebra of $\mathfrak{Re}(U)$ with atoms 1' and 0' ($\mathfrak{A}$ is isomorphic to $\mathfrak{C}_1[\langle 0\rangle]$), and let $M = \begin{pmatrix} 1' & 0' & 0' & 0' \\ 0' & 1' & 0' & 0' \\ 0' & 0' & 1' & 0' \\ 0' & 0' & 0' & 1' \end{pmatrix}$. Then M is a proto-solution of itself, but it has no solution, since any solution of M must be a quadruple $\langle p_1,p_2,p_3,p_4\rangle$ with distinct entries, but there are only three elements in U. On the other hand, M can be considered as a $\mathfrak{C}_1[\langle 0\rangle]$-matrix, in which case it does have solutions, namely all quadruples of distinct real numbers.

We have seen that proto-solutions can exist when solutions do not. It is also possible for solutions to exist when proto-solutions do not: an infinite atomic subalgebra $\mathfrak{A}$ of $\mathfrak{Re}(U)$, where U is a countable infinite set, and an $\mathfrak{A}$-matrix M with a solution but no proto-solution over $\mathfrak{A}$.

Examples of this are more difficult to construct, but can be found in [22] and [25]. For such an example, however, it is necessary that $\mathfrak{A}$ be infinite [20, Theorem 5.7].

For the IA, the situation is quite nice. An IA-matrix has a solution if and only if it has a proto-solution [21], [20], and this is true for all isomorphic copies of the IA which are embedded in algebras $\mathfrak{Re}(U)$ where U is not necessarily the set of events based on real numbers. Constraint satisfiability for the IA is NP-complete. A sketch of a proof of this was given in [51]. The idea of that proof is to reduce the 3-clause satisfiability problem (for propositional calculus) to constraint satisfiability for the IA. (Additional details for that proof are given in [52].) Another proof is sketched in [48]. where graph-colorability is reduced to constraint satisfiability for the IA. Both of these proofs deal with solutions, not proto-solutions, but, in view of the remarks made above, this makes no difference to the IA.

The NP-completeness of the constraint satisfiability problem for the IA follows from Theorem 2 below. This theorem is not restricted to the IA and indeed applies some compass algebras. It also applies to infinite algebras, such as the Allen-Hayes algebra, and to nonrepresentable algebras.

Theorem 2. *Assume $\mathfrak{A}$ is a relation algebra with elements $x, y, z \neq 0$, such that*

(i) $1', x, \breve{x}, y, \breve{y}, z, \breve{z}$ *are pairwise disjoint,*
(ii) $z \cdot z;y = 0$,
(iii) $y \cdot x;y = 0$,
(iv) $x \cdot x;y = 0$,
(v) $z \cdot x;z = 0$,
(vi) $y \cdot y;z = 0$,
(vii) $x \cdot z;x = 0$,
(viii) $z \leq x;y$, $x \leq z;\breve{y}$, $y \leq \breve{x};z$,
(ix) $1' \leq x;\breve{x} \cdot \breve{x};x \cdot y;\breve{y} \cdot \breve{y};y \cdot z;\breve{z} \cdot \breve{z};z$.

Then following problem is NP-complete: **(R)** *Determine whether a matrix M over $\mathfrak{A}$ has a closed zeroless reduction bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$.*

Proof. It suffices to show that Graph 3-Colorability [10] is reducible to (R). Let $G = \langle V, E\rangle$ be a graph (*i.e.*, E is a symmetric binary relation on V that is disjoint from the identity relation on V). We may assume without loss of generality that the set V of vertices of G is $\{4, \ldots, |V| + 3\}$, where $|V|$ is the cardinality of V. Let $n = |V| + 3$. Let M be the n-by-n $\mathfrak{A}$-matrix determined by the following stipulations:

(i) $M_{ii} = 1'$ for $1 \leq i \leq n$,
(ii) $M_{12} = x$, $M_{21} = \breve{x}$, $M_{23} = y$, $M_{32} = \breve{y}$, $M_{13} = z$, $M_{31} = \breve{z}$,
(iii) $M_{i1} = 1' + \breve{x} + \breve{z}$, $M_{1i} = 1' + x + z$, $M_{i2} = 1' + x + \breve{y}$, $M_{2i} = 1' + \breve{x} + y$, $M_{i3} = 1' + y + z$, and $M_{3i} = 1' + \breve{y} + \breve{z}$ whenever $i \in V$ (*i.e.*, $4 \leq i \leq n$),
(iv) $M_{ij} = M_{ji} = x + \breve{x} + y + \breve{y} + z + \breve{z}$ whenever $i, j \in V$ and $\langle i, j\rangle \in E$,
(v) in all other cases, $M_{ij} = 1$.

We will show that there is a natural one-to-one correspondence between 3-colorings of the graph G and closed zeroless reductions of M which are bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$. It follows that M has a closed zeroless reduction bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$ just in case the graph G is 3-colorable.

Suppose that N is a closed zeroless reduction of M which is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$. We will show that N determines a 3-coloring $\gamma : V \to \{1, 2, 3\}$ of G. First, since $1', x, \breve{x}, y, \breve{y}, z, \breve{z}$ are pairwise disjoint, N is zeroless, and N is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$, we concluded that if $1 \leq i, j \leq n$, then exactly one of the following seven statements holds: $N_{ij} \leq 1'$, $N_{ij} \leq x$, $N_{ij} \leq \breve{x}$, $N_{ij} \leq y$, $N_{ij} \leq \breve{y}$, $N_{ij} \leq z$, $N_{ij} \leq \breve{z}$. Now we look at the possible values of N_{i1}, N_{i2}, and N_{i3} for an arbitrary $i \in V$, *i.e.*, for $4 \leq i \leq n$. Since $N \leq M$, we have

$$N_{12} \leq x, \ N_{23} \leq y, \ N_{13} \leq z,$$
$$N_{i1} \leq 1' + \breve{x} + \breve{z}, \ N_{i2} \leq 1' + x + \breve{y}, \ N_{i3} \leq 1' + y + z.$$

If $N_{i1} \leq 1'$, then

$$N_{i2} \leq N_{i1};N_{12} \leq 1';x = x,$$
$$N_{i3} \leq N_{i1};N_{13} \leq 1';z = z.$$

Similarly, if $N_{i2} \leq 1'$, then

$$N_{i1} \leq N_{i2};N_{21} \leq 1';\breve{x} = \breve{x},$$
$$N_{i3} \leq N_{i2};N_{23} \leq 1';y = y.$$

Finally, if $N_{i3} \leq 1'$, then

$$N_{i1} \leq N_{i3};N_{31} \leq 1';\breve{z} = \breve{z},$$
$$N_{i2} \leq N_{i3};N_{32} \leq 1';\breve{y} = \breve{y}.$$

From these observations it follows that $N_{ik} \leq 1'$ for at most one $k \in \{1, 2, 3\}$. To show $N_{ik} \leq 1'$ for at least one $k \in \{1, 2, 3\}$, we assume $N_{i1} \leq \breve{x} + \breve{z}$, $N_{i2} \leq x + \breve{y}$, and $N_{i3} \leq y + z$, and derive a contradiction. There are two cases. First, if $N_{i3} \leq y$, then

$$N_{i1} \leq (\breve{x} + \breve{z}) \cdot N_{i3};N_{31} \leq (\breve{x} + \breve{z}) \cdot y;\breve{z} \leq \breve{x},$$
$$N_{i2} \leq (x + \breve{y}) \cdot N_{i3};N_{32} \leq (x + \breve{y}) \cdot y;\breve{y} \leq \breve{y}$$

by (ii) and (iii), respectively. From these last two equations we get

$$N_{i2} \leq \breve{y} \cdot N_{i1};N_{12} \leq \breve{y} \cdot \breve{x};x = 0$$

by (iv), contradicting the assumption that N is zeroless. Second, if $N_{i3} \leq z$, then

$$N_{i1} \leq (\breve{x} + \breve{z}) \cdot N_{i3};N_{31} \leq (\breve{x} + \breve{z}) \cdot z;\breve{z} \leq \breve{z},$$
$$N_{i2} \leq (x + \breve{y}) \cdot N_{i3};N_{32} \leq (x + \breve{y}) \cdot z;\breve{y} \leq x$$

by (v) and (vi), respectively. From these last two equations we get

$$N_{i2} \leq x \cdot N_{i1};N_{12} \leq x \cdot \breve{z};x = 0$$

by (vii), again contradicting the assumption that N is zeroless. This exhausts the possibilities. Thus we have $N_{ik} \leq 1'$ for exactly one $k \in \{1, 2, 3\}$. This allows us to define $\gamma : V \rightarrow \{1, 2, 3\}$ by $\gamma(i) = k$ iff $N_{ik} \leq 1'$, for every $i \in V$. Now if $\langle i, j \rangle \in E$, then we must have $\gamma(i) \neq \gamma(j)$, for if $\gamma(i) = \gamma(j) = k$, then we have $N_{ik} \leq 1'$ and $N_{jk} \leq 1'$, from which we obtain

$$x + \breve{x} + y + \breve{y} + z + \breve{z} = N_{ij} \leq N_{ik};N_{kj} \leq 1';\breve{1}' = 1',$$

contradicting (i). Thus γ is a 3-coloring of G.

For the other direction, if we have a 3-coloring $\gamma : V \rightarrow \{1, 2, 3\}$ of G, we can get a closed zeroless reduction $N \leq M$ which is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$ as follows. Set $N_{12} = x$, $N_{21} = \breve{x}$, $N_{23} = y$, $N_{32} = \breve{y}$, $N_{13} = z$, $N_{31} = \breve{z}$, and $N_{ii} = 1'$ whenever $1 \leq i \leq n$. For all $i, j \in V$, and every $k \in \{1, 2, 3\}$, set $N_{ik} = N_{\gamma(i)k}$, $N_{ki} = N_{k\gamma(i)}$, and $N_{ij} = N_{\gamma(i)\gamma(j)}$. It follows from (viii) and (ix) that this definition gives a closed zeroless matrix N. Obviously, N is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$. The fact that γ is a 3-coloring of G is used to show that $N \leq M$. □

Corollary 3. (i) *Constraint satisfiability for $C_2[\langle 1,0 \rangle, \langle 1,1 \rangle, \langle 0,1 \rangle]$ is NP-complete. The same is true for the symmetric subalgebra of $C_2[\langle 1,0 \rangle, \langle 1,1 \rangle, \langle 0,1 \rangle]$.*

(ii) *Constraint satisfiability for the the IA is NP-complete. The same is true for the Allen-Hayes algebra.*

Proof. (i): Use Theorem 2 with $x = \hat{a}$, $y = \hat{c}$, and $z = \hat{e}$.

(ii): Use Theorem 2 with $x = m$, $y = f$, and $z = s$. □

Theorem 2 applies to a 3-directional compass algebra. For 2-directional compass algebras we need another theorem.

Theorem 4. *Let $\mathfrak{A}$ be a relation algebra with nonsero elements x, y, z such that*

(i) $1', x, \breve{x}, y, \breve{y}, z, \breve{z}$ *are pairwise disjoint,*
(ii) $y \cdot x;x = 0$,
(iii) $y \cdot x;y = 0$,
(iv) $y \cdot y;x = 0$,
(v) $y \cdot y;z = 0$,
(vi) $y \cdot z;y = 0$,
(vii) $y \cdot z;z = 0$,
(viii) $x \cdot z;x = 0$,
(ix) $x \quad x;z = 0$,
(x) $z \leq z;z$, $z \leq z;\breve{z}$, $z \leq \breve{z};z$,
(xi) $y \leq y;y$, $y \leq y;\breve{y}$, $y \leq \breve{y};y$,
(xii) $y \leq x;z$, $x \leq y;\breve{z}$, $z \leq \breve{x};y$,
(xiii) $y \leq z;x$, $z \leq y;\breve{x}$, $x \leq \breve{z};y$,
(xiv) $z \leq z;y$, $z \leq z;\breve{y}$, $y \leq \breve{z};z$,
(xv) $z \leq y;z$, $y \leq z;\breve{z}$, $z \leq \breve{y};z$,
(xvi) $x \leq z;x$, $z \leq z;\breve{x}$, $x \leq \breve{z};z$,
(xvii) $z \leq x;z$, $x \leq z;\breve{z}$, $z \leq \breve{x};z$,
(xviii) $1' \leq x;\breve{x} \cdot \breve{x};x \cdot y;\breve{y} \cdot \breve{y};y \cdot z;\breve{z} \cdot \breve{z};z$.

Then the following problem is NP-complete: (**R**) *Determine whether a network N over $\mathfrak{A}$ with labels in $\{y, x+y+z, z+\breve{z}\}$ has a closed seroless reduction bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$.*

Proof. As in the previous proof, we show that Graph 3-Colorability [10] is reducible to (**R**). Let $G = \langle V, E\rangle$ be a graph with vertex set $V = \{3, \ldots, |V|+2\}$, Let $n = |V|+2$. Let M be the n-by-n $\mathfrak{A}$-matrix determined by the following stipulations: $M_{12} = y$, $M_{21} = \breve{y}$, $M_{1i} = M_{i2} = x+y+z$ for every $i \in V$, $M_{ij} = z + \breve{z}$ whenever $i, j \in V$ and $\langle i, j\rangle \in E$, The 3-colorings of G correspond to closed seroless reductions of M which are bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$.

Suppose that N is a closed seroless reduction of M which is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$. We show that N determines a 3-coloring $\gamma : V \to \{1,2,3\}$. First, if $1 \leq i, j \leq n$, then exactly one of the following seven statements holds: $N_{ij} \leq 1'$, $N_{ij} \leq x$, $N_{ij} \leq \breve{x}$, $N_{ij} \leq y$, $N_{ij} \leq \breve{y}$, $N_{ij} \leq z$, and $N_{ij} \leq \breve{z}$. Now we look at the possible values of N_{1i} and N_{i2} for an arbitrary $i \in V$. We have $N_{12} \leq y$, $N_{1i} \leq x+y+z$, $N_{i2} \leq x+y+z$. Hence there are nine cases, six of which are ruled out because they contradict one of the hypotheses. For example, if $N_{1i} \leq x$ and $N_{i2} \leq y$, then by (iii) we have

$$N_{12} \leq y \cdot N_{1i};N_{i2} \leq y \cdot x;y = 0$$

contradicting the assumption that N is seroless. The following table shows which cases are ruled out by hypotheses (ii)–(vii).

	$N_{i2} \leq x$	$N_{i2} \leq y$	$N_{i2} \leq z$
$N_{1i} \leq x$	No, by (ii).	No, by (iii).	
$N_{1i} \leq y$	No, by (iv).		No, by (v).
$N_{1i} \leq z$		No, by (vi).	No, by (vii).

The remaining three cases are used to define $\gamma : V \to \{1,2,3\}$. For every $i \in V$,

$$\gamma(i) = \begin{cases} 1 & \text{if } N_{1i} \leq z \text{ and } N_{i2} \leq x \\ 2 & \text{if } N_{1i} \leq y \text{ and } N_{i2} \leq y \\ 3 & \text{if } N_{1i} \leq x \text{ and } N_{i2} \leq z \end{cases}.$$

Now we must show $\gamma(i) \neq \gamma(j)$ whenever $\langle i, j\rangle \in E$. Since N is closed and $N \leq M$,

$$N_{ij} \leq z + \breve{z}, \quad N_{ji} \leq z + \breve{z}.$$

If $\gamma(i) = 1$ then $N_{i2} \leq x$, so by (ix) we get

$$\begin{aligned} N_{j2} &\leq (x+y+z) \cdot N_{ji};N_{i2} \\ &\leq (x+y+z) \cdot (z+\breve{z});x \\ &\leq (x+y+z) \cdot (z;x+\breve{z};x) \leq y+z. \end{aligned}$$

Therefore, either $N_{j2} \leq y$ and $\gamma(j) = 2$, or else $N_{j2} \leq z$ and $\gamma(j) = 3$. Thus $\gamma(i) \neq \gamma(j)$. If $\gamma(i) = 2$ then $N_{i2} \leq y$, so

$$\begin{aligned} N_{j2} &\leq (x+y+z) \cdot N_{ji};N_{i2} \\ &\leq (x+y+z) \cdot (z+\breve{z});y \\ &\leq (x+y+z) \cdot (z;y+\breve{z};y) \leq x+z. \end{aligned}$$

by (vi). Thus either $N_{j2} \leq x$ and $\gamma(j) = 1$, or else $N_{j2} \leq z$ and $\gamma(j) = 3$. Again, $\gamma(i) \neq z(j)$. Finally, if $\gamma(i) = 3$ then $N_{1i} \leq x$, so

$$\begin{aligned} N_{1j} &\leq (x+y+z) \cdot N_{1i};N_{ij} \\ &\leq (x+y+z) \cdot x;(z+\breve{z}) \\ &\leq (x+y+z) \cdot (x;z+x;\breve{z}) \leq y+z. \end{aligned}$$

by (viii). Either $N_{j2} \leq x$ and $\gamma(j) = 2$, or else $N_{j2} \leq z$ and $\gamma(j) = 1$. Hence $\gamma(i) \neq \gamma(j)$. This completes the proof that $\gamma(i) \neq \gamma(j)$ whenever $\langle i,j \rangle \in E$, and shows that γ is a 3-coloring of G.

For the other direction, if we have a 3-coloring $\gamma : V \rightarrow \{1,2,3\}$, we can get a closed zeroless reduction $N \leq M$ which is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$. Set $N_{12} = y$, $N_{21} = \breve{y}$, and $N_{ii} = 1'$ whenever $1 \leq i \leq n$. For all $i, j \in V$, set

$$N_{ij} = \begin{cases} 1' & \text{if } \gamma(i) = \gamma(j) \\ z & \text{if } \gamma(i) > \gamma(j) \\ \breve{z} & \text{if } \gamma(i) < \gamma(j) \end{cases},$$

$$N_{1i} = \begin{cases} z & \text{if } \gamma(i) = 1 \\ y & \text{if } \gamma(i) = 2 \\ x & \text{if } \gamma(i) = 3 \end{cases}, \quad N_{i2} = \begin{cases} x & \text{if } \gamma(i) = 1 \\ y & \text{if } \gamma(i) = 2 \\ z & \text{if } \gamma(i) = 3 \end{cases},$$

$$N_{i1} = \begin{cases} \breve{z} & \text{if } \gamma(i) = 1 \\ \breve{y} & \text{if } \gamma(i) = 2 \\ \breve{x} & \text{if } \gamma(i) = 3 \end{cases}, \quad N_{2i} = \begin{cases} \breve{x} & \text{if } \gamma(i) = 1 \\ \breve{y} & \text{if } \gamma(i) = 2 \\ \breve{z} & \text{if } \gamma(i) = 3 \end{cases}.$$

It follows from (x)–(xviii) that N is closed and zeroless. Obviously, N is bounded by $\{1', x, \breve{x}, y, \breve{y}, z, \breve{z}\}$. The fact that γ is a 3-coloring of G is used to show that $N \leq M$. □

To see the necessity of (x)–(xviii), consider the following example. Let $G = \langle V, E \rangle$ where $V = \{3,4,5\}$ and $E = \emptyset$. Let $\gamma : V \rightarrow \{1,2,3\}$ be the 3-coloring of G defined by $\gamma(i) = i - 2$ for every $i \in V$. The resulting N is shown below. The matrix N is closed iff (x)–(xviii) hold.

$$N = \begin{pmatrix} 1' & y & z & y & x \\ \breve{y} & 1' & \breve{x} & \breve{y} & \breve{z} \\ \breve{z} & x & 1' & \breve{z} & \breve{z} \\ \breve{y} & y & z & 1' & \breve{z} \\ \breve{x} & z & z & z & 1' \end{pmatrix}$$

Corollary 5. *Constraint satisfiability for $C_2[\langle 1,0 \rangle, \langle 0,1 \rangle]$ is NP-complete. The same is true for any compass algebra with at least two directions.*

Proof. By Theorem 4, with $x = a$, $y = c$, and $z = d$. □

These theorems can be extended to show that essentially all but the most trivial compass and interval algebras have NP-hard constraint satisfaction problems.

6. Relational semantics

The results in this section are stated without proofs. For proofs and additional details see [23] and [24].

Let $\mathcal{L}$ be a programming language which contains two disjoint classes of objects $\mathcal{L}_P$ and $\mathcal{L}_C$, called the *predicates* and *commands* of $\mathcal{L}$, respectively. The commands are of two types, basic and compound. Among the basic commands are havoc, abort, and skip. There may be other basic commands, *e.g.*, assignment statements, but they will not be treated here. The compound commands are closed under three formation rules, and every compound command can be obtained in exactly one of these three ways.

(i) If S_0, S_1 are commands then so is $S_0;S_1$.
(ii) If S is a command and B is a predicate, then do $B \rightarrow S$ od is a command.
(iii) If $\{S_i : i \in I\}$ is a set of commands and $\{B_i : i \in I\}$ is a set of predicates, then if $i{:}B_i \rightarrow S_i$ fi is a command.

If $\{S_i : i \in I\} = \{S\}$ and $\{B_i : i \in I\} = \{B\}$ we denote if $i{:}B_i \rightarrow S_i$ fi by simply if $B \rightarrow S$ fi.

Imagine that U is a set of machine states, that each command S has an associated "input-output relation" $\mathfrak{r}_S$ containing all pairs of states $\langle p, q\rangle$ for which there is a terminating computation of S starting at input state p and ending at output state q, that each command S has a "nontermination relation" $\mathfrak{e}_S$ of the form $E \times U$, where E is the set of states initiating nonterminating (or "eternal") computations of S, and that each predicate B has a corresponding relation $\mathfrak{d}_B$ of the form $X \times U$, where X is the set of states satisfying B. An element x of a relation algebra is *domain element* is $x;1 = x$. Thus $\mathfrak{e}_S$ and $\mathfrak{d}_B$ are domain elements of $\mathfrak{Re}(U)$, and $\mathfrak{Re}(U), \mathfrak{r}, \mathfrak{e}, \mathfrak{d}$ is concrete example of an "interpretation", called an "operational interpretation". The concept of interpretation is generalized from this concrete case and defined for an arbitrary relation algebra as follows.

Definition 6. *An interpretation of $\mathcal{L}$ is a relation algebra $\mathfrak{A} = \langle A, +, \cdot, ^-, 0, 1, \dagger, ;, ^\smile, 0', 1'\rangle$ together with three maps*

$$\mathfrak{r} : \mathcal{L}_C \rightarrow A, \quad \mathfrak{e} : \mathcal{L}_C \rightarrow A, \quad \textit{and} \quad \mathfrak{d} : \mathcal{L}_P \rightarrow A,$$

such that

(i) $\mathfrak{e}_S;1 = \mathfrak{e}_S$ *for every command* $S \in \mathcal{L}_C$,
(ii) $\mathfrak{d}_B;1 = \mathfrak{d}_B$ *for every predicate* $B \in \mathcal{L}_P$.

Each command S has its associated "weakest-liberal-precondition" and "weakest-precondition" transformers (and their duals), defined by

(i) $\mathrm{wlp}_S(x) = \overline{\mathfrak{r}_S;\overline{x}} = \overline{\mathfrak{r}_S} \dagger x$,
(ii) $\mathrm{wp}_S(x) = \overline{\mathfrak{r}_S;\overline{x}} \cdot \overline{\mathfrak{e}_S} = (\overline{\mathfrak{r}_S} \dagger x) \cdot \overline{\mathfrak{e}_S}$,
(iii) $\mathrm{wlp}^*_S(x) = \overline{\mathrm{wlp}_S(\overline{x})} = \mathfrak{r}_S;x$,
(iv) $\mathrm{wp}^*_S(x) = \overline{\mathrm{wp}_S(\overline{x})} = \mathfrak{r}_S;x + \mathfrak{e}_S$.

In case x is a domain element, $\mathrm{wlp}_S(x)$ is called the "weakest liberal precondition guaranteeing x", and $\mathrm{wp}_S(x)$ is called the "weakest precondition guaranteeing x".

Theorem 7. *If x is a domain element, then $\mathrm{wlp}_S(x)$ and $\mathrm{wp}_S(x)$ are also domain elements.*

We will usually apply the functions $\mathrm{wlp}_S(\text{-})$ and $\mathrm{wp}_S(\text{-})$ only to domain elements, although they are defined for all elements of the relation algebra $\mathfrak{A}$. The extended definition allows the recovery of $\mathfrak{r}_S$ from $\mathrm{wlp}_S(\text{-})$, since $\mathfrak{r}_S = \mathrm{wlp}^*_S(1')$. The extended definition allows something more. Suppose we consider two commands S_0, S_1, and we wish to construct from them a command S_2 such that $\mathfrak{r}_{S_2};\mathfrak{r}_{S_1} \leq \mathfrak{r}_{S_0}$. According to De Morgan's Theorem K, this condition is equivalent to $\mathfrak{r}_{S_2} \leq \overline{\overline{\mathfrak{r}_{S_0}};\breve{\mathfrak{r}}_{S_1}}$, but $\overline{\overline{\mathfrak{r}_{S_0}};\breve{\mathfrak{r}}_{S_1}} = (\mathrm{wlp}_{S_1}(\breve{\mathfrak{r}}_{S_0}))^\smile$, so we can use any S_2 such that $\mathfrak{r}_{S_2} \leq (\mathrm{wlp}_{S_1}(\breve{\mathfrak{r}}_{S_0}))^\smile$.

The relation $\overline{\overline{\mathrm{r}_{S_0}};\breve{\mathrm{r}}_{S_1}}$ is called the "weakest prespecification" of S_0 and S_1 [13, p. 684]. The weakest prespecification was explicitly mentioned by Peirce in [38] (under a different name, of course). The converse-dual of the weakest prespecification, namely $\overline{\overline{x};y}$, was already introduced by De Morgan in [35] and called "progressive involution" by Peirce. Many algebraic laws governing this operation can be found in [42], and some of them are proved in [14] and [15] from a different axiomatisation for relation algebras.

Theorem 8. *The following laws hold for arbitrary interpretations.*

(i) $\mathrm{wp}_S(x) = \mathrm{wlp}_S(x) \cdot \overline{\mathrm{e}_S}$,
(ii) $\mathrm{wlp}_S(1) = 1$,
(iii) $\mathrm{wp}_S(1) = \overline{\mathrm{e}_S}$,
(iv) $\mathrm{wp}_S(x) = \mathrm{wlp}_S(x) \cdot \mathrm{wp}_S(1)$,
(v) $\mathrm{r}_S = \mathrm{wlp}^*_S(1')$,
(vi) $\mathrm{e}_S = \mathrm{wp}^*_S(0)$,
(vii) $\mathrm{wlp}^*_S(\text{-})$, $\mathrm{wlp}_S(\text{-})$, $\mathrm{wp}^*_S(\text{-})$, and $\mathrm{wp}_S(\text{-})$ *are monotone (preserve inclusions),*
(viii) $\mathrm{wlp}^*_S(\text{-})$ *distributes over arbitrary joins,*
(ix) $\mathrm{wlp}_S(\text{-})$ *distributes over arbitrary meets,*
(x) $\mathrm{wp}^*_S(\text{-})$ *distributes over nonempty joins,*
(xi) $\mathrm{wp}_S(\text{-})$ *distributes over nonempty meets,*
(xii) $\mathrm{wlp}_S(x) \cdot \mathrm{wlp}_S(y) = \mathrm{wlp}_S(x \cdot y)$,
(xiii) $\mathrm{wp}_S(x) \cdot \mathrm{wp}_S(y) = \mathrm{wp}_S(x \cdot y)$,
(xiv) $\mathrm{wp}_S(x) \cdot \mathrm{wlp}_S(y) = \mathrm{wp}_S(x \cdot y)$,
(xv) *If* $\mathrm{wp}_S(0) = 0$ *then* $\mathrm{wp}_S(x) \leq \mathrm{wlp}^*_S(x)$.

Definition 9 below is based on the remarks in [9, p. 137]. What is actually used as a definition of "S is deterministic" in [9] depends on the assumption that $\mathrm{wp}_S(0) = 0$, and appears in Theorem 10. Determinism in the arbitrary case is characterised in Theorem 11, which says that S is deterministic if and only if r_S is a partial function, and no state initiates both a terminating and a nonterminating computation of S, *i.e.*, r_S and e_S have disjoint domains. Note that a deterministic S can still have nonterminating computations.

Definition 9. *A command* $S \in \mathcal{L}_C$ *is* deterministic *if* $\mathrm{wlp}^*_S(x) \leq \mathrm{wp}_S(x)$ *for all* x.

Theorem 10. *If* $\mathrm{wp}_S(0) = 0$, *then S is deterministic iff* $\mathrm{wlp}^*_S(x) = \mathrm{wp}_S(x)$ *for all* x.

Theorem 11. *A command S is deterministic iff* $\breve{\mathrm{r}}_S;\mathrm{r}_S \leq 1'$ *and* $\mathrm{r}_S \cdot \mathrm{e}_S = 0$.

Now we turn to the definition of a "correct" interpretation, one which respects the intended meanings of the basic commands and command structures given above. The remarks following Definition 12 are justifications for the correspondingly labeled parts of Definition 12. In motivating the definition of correct interpretation we freely form joins which, in case we are dealing with an algebra $\mathfrak{Re}(U)$ of all binary relations on the universe of states U, are simply unions and certainly do exist. In the abstract definition, however, we need to know that various joins exist, and so, in order to avoid lengthy formulations of results, we ask that the relation algebra used in a correct interpretation be complete.

Definition 12. *An interpretation is* correct *if* $\mathfrak{A}$ *is complete and the following conditions hold.*

(i) $\mathrm{r}_{\mathbf{havoc}} = 1$ *and* $\mathrm{e}_{\mathbf{havoc}} = 0$.
(ii) $\mathrm{r}_{\mathbf{abort}} = 0$ and $\mathrm{e}_{\mathbf{abort}} = 1$.
(iii) $\mathrm{r}_{\mathbf{skip}} = 1'$ and $\mathrm{e}_{\mathbf{skip}} = 0$.
(iv) *For all commands* S_0, S_1, $\mathrm{r}_{S_0;S_1} = \mathrm{r}_{S_0};\mathrm{r}_{S_1}$ and $\mathrm{e}_{S_0;S_1} = \mathrm{e}_{S_0} + \mathrm{r}_{S_0};\mathrm{e}_{S_1}$.
(v) *For all I-indexed sets* $\{B_i : i \in I\}$ *of predicates and* $\{S_i : i \in I\}$ *of commands,*

$$\mathrm{r}_{\mathbf{if}\, i:B_i \to S_i\, \mathbf{fi}} = \sum_{i\in I} (\mathrm{d}_{B_i} \cdot \mathrm{r}_{S_i}), \qquad \mathrm{e}_{\mathbf{if}\, i:B_i \to S_i\, \mathbf{fi}} = \prod_{i\in I} \overline{\mathrm{d}_{B_i}} + \sum_{i\in I} (\mathrm{d}_{B_i} \cdot \mathrm{e}_{S_i}).$$

(vi) *For every predicate B and every command S,*

$$\mathrm{r}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}} = \sum_{i\in\omega}\left((\mathrm{d}_B\cdot\mathrm{r}_S)^i\,;\,(\overline{\mathrm{d}_B}\cdot 1')\right),\qquad \mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}} = \sum\{y : y\le \mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S\,;y)\}.$$

Remarks on parts of Definition 12:

(i) Every execution of havoc terminates; upon termination the machine may be in any state. Thus every state is connected to every other state by a terminating computation of havoc, and havoc has no nonterminating computations.

(ii) For every initial state the execution of abort fails to terminate, that is, every state initiates a nonterminating computation of abort, and abort has no terminating computations.

(iii) Every execution of skip is guaranteed to terminate and leaves the state of the machine unchanged, that is, there are no nonterminating computations, and every computation has the same final state as initial state.

(iv) The operational interpretation of $S_0;S_1$ is "first execute S_0, then execute S_1". Thus a terminating computation of $S_0;S_1$ starts at a state that begins a terminating computation of S_0 that ends at a state that begins a terminating computation of S_1 that ends at the final state of the computation of $S_0;S_1$. A state initiates a nonterminating computation of $S_0;S_1$ if it either initiates a nonterminating computation of S_0, or else initiates a terminating computation of S_0 that ends at a state that begins a nonterminating computation of S_1.

(v) A computation is a terminating computation of if $i{:}B_i\rightarrow S_i$ fi if, for some $i\in I$, it is a terminating computation of S_i whose initial state satisfies B_i. The states initiating nonterminating computations of if $i{:}B_i\rightarrow S_i$ fi are those in which no B_i is satisfied, together with those which, for some $i\in I$, satisfy B_i and initiate a nonterminating computation of S_i.

(vi) A terminating computation for do $B\rightarrow S$ od is a finite sequence (possibly empty) of terminating computations of if $B\rightarrow S$ fi, such that the last computation terminates at a state not satisfying B. Consider a state p from which a nonterminating computation of *do $B\rightarrow S$ od* is possible. First, B must hold at p, since otherwise the execution of do $B\rightarrow S$ od would terminate immediately. Therefore p in the domain of d_B. Since B holds, S is executed. This either leads to a nonterminating computation of S, that is, p is in the domain of e_S, or else there is no such nonterminating computation. Therefore p must initiate a terminating computation of S, for if not, we would have a state satisfying B from which no computation of S is possible, contradicting our assumption that p does initiate a computation of do $B\rightarrow S$ od. Thus p initiates no nonterminating computations of S, but does initiate a nonterminating computation of do $B\rightarrow S$ od, so at least one of the terminating computations of S must end in a state from which a nonterminating computation of do $B\rightarrow S$ od is possible. This conclusion is equivalent to asserting that p is in the domain of $\mathrm{r}_S;\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$. Putting these inclusions together, we conclude that any state in the domain of $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$ must be in the domain of $\mathrm{d}_B\cdot\mathrm{e}_S+\mathrm{r}_S;\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$, that is, $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}\le\mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S;\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}})$. Thus $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$ is a solution of $y\le\mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S;y)$. Conversely, we can argue that if $y\le\mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S;y)$ then $y\le\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$. Indeed, a state p in the domain of y must satisfy B, and either a nonterminating computation of S is possible from p, in which case p initiates a nonterminating computation of do $B\rightarrow S$ od, or else p initiates a terminating computation of S that ends in a state p' which is again in the domain of y. Either p' initiates a nonterminating computation of S or a terminating computation of S that ends at a state p'' in the domain of y, and so on. We either eventually get into a nonterminating computation of S, or else create an infinite sequence of terminating computations of S. Either way we get a nonterminating computation of do $B\rightarrow S$ od, so p is in the domain of $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$. Since $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$ is a domain relation, this argument is enough to show $y\le\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$. Thus $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$ is, in fact, the largest solution of $y\le\mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S;y)$. Let $f(y)=\mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S;(y))$. Then f is monotone, so by Tarski's Fixed Point Theorem [45], the largest solution of $y\le f(y)$ is $\sum\{y : y\le f(y)\}$. We therefore set $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}=\sum\{y : y\le f(y)\}$ in Definition 12.

Incidentally, Tarski's Fixed Point Theorem [45] also asserts that $\sum\{y : y\le f(y)\}=\sum\{y : y=f(y)\}$, so $\mathrm{e}_{\mathrm{do}\,B\rightarrow S\,\mathrm{od}}$ is the largest fixed point of $\mathrm{d}_B\cdot(\mathrm{e}_S+\mathrm{r}_S;(\text{-}))$. We can also express

$\mathrm{r}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}}$ is the smallest fixed point of the function $\overline{\mathrm{d}_B} \cdot 1' + \mathrm{d}_B \cdot \mathrm{r}_S ; (\text{-})$, *i.e.*, $\mathrm{r}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}} = \prod\{y : \overline{\mathrm{d}_B} \cdot 1' + \mathrm{d}_B \cdot \mathrm{r}_S ; y \leq y\}$.

Definition 12 is concerned only with those language features used here. For our present purposes, the predicates of $\mathcal{L}$ need only form a nonempty set, but if the predicates of $\mathcal{L}$ contain constants true, false, and are closed under standard connectives of propositional calculus, then the following conditions could be added to the definition of correctness.

$$\mathrm{d}_{\mathsf{true}} = 1 \qquad \mathrm{d}_{\mathsf{false}} = 0$$
$$\mathrm{d}_{\neg B} = \overline{\mathrm{d}_B} \qquad \mathrm{d}_{B\wedge C} = \mathrm{d}_B \cdot \mathrm{d}_C$$
$$\mathrm{d}_{B\vee C} = \mathrm{d}_B + \mathrm{d}_C \qquad \mathrm{d}_{B\rightarrow C} = \overline{\mathrm{d}_B} + \mathrm{d}_C$$
$$\mathrm{d}_{B\leftrightarrow C} = \mathrm{d}_B \cdot \mathrm{d}_C + \overline{\mathrm{d}_B} \cdot \overline{\mathrm{d}_C}$$

Correct interpretations are extremely abundant.

Theorem 13. *For every language $\mathcal{L}$ and every complete relation algebra $\mathfrak{A}$, if we assume that*

(i) d *is any map from predicates to domain elements of* $\mathfrak{A}$,

(ii) r' *is any map from basic commands to elements of* $\mathfrak{A}$ *such that* $r'(\mathsf{havoc}) = 1$, $r'(\mathsf{abort}) = 0$, *and* $r'(\mathsf{skip}) = 1'$,

(iii) e' *is a any map from basic commands to domain elements of* $\mathfrak{A}$ *such that* $e'(\mathsf{havoc}) = 0$, $e'(\mathsf{abort}) = 1$, *and* $e'(\mathsf{skip}) = 0$,

then r' *and* e' *can be extended in a unique way to maps* r *and* e *such that* $\mathfrak{A}, \mathrm{r}, \mathrm{e}, \mathrm{d}$ *is a correct interpretation.*

Theorem 14. *The following laws hold for an arbitrary correct interpretation of* $\mathcal{L}$.

(i) $\mathrm{wlp}_{\mathsf{havoc}}(x) = 0\dagger x$, $\mathrm{wp}_{\mathsf{havoc}}(x) = 0\dagger x$,

(ii) $\mathrm{wlp}_{\mathsf{abort}}(x) = 1$, $\mathrm{wp}_{\mathsf{abort}}(x) = 0$,

(iii) $\mathrm{wlp}_{\mathsf{skip}}(x) = x$, $\mathrm{wp}_{\mathsf{skip}}(x) = x$,

(iv) $\mathrm{wlp}_{S_0;S_1}(x) = \mathrm{wlp}_{S_0}(\mathrm{wlp}_{S_1}(x))$,

(v) $\mathrm{wp}_{S_0;S_1}(x) = \mathrm{wp}_{S_0}(\mathrm{wp}_{S_1}(x))$,

(vi) $\mathrm{wlp}_{\mathsf{if}\, i:B_i\rightarrow S_i\, \mathsf{fi}}(x) = \prod\limits_{i\in I}(\overline{\mathrm{d}_{B_i}} + \mathrm{wlp}_{S_i}(x))$,

(vii) $\mathrm{wp}_{\mathsf{if}\, i:B_i\rightarrow S_i\, \mathsf{fi}}(x) = \prod\limits_{i\in I}(\overline{\mathrm{d}_{B_i}} + \mathrm{wp}_{S_i}(x)) \cdot \sum\limits_{i\in I}\mathrm{d}_{B_i}$,

(viii) $\mathrm{wlp}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}}(x) = \overline{\sum\limits_{i\in\omega}\left((\mathrm{d}_B \cdot \mathrm{r}_S)^i ; \overline{\mathrm{d}_B} \cdot \overline{x}\right)} = \prod\limits_{i\in\omega}(\mathrm{wlp}_{\mathsf{if}\, B\rightarrow S\, \mathsf{fi}})^i(\mathrm{d}_B + x)$,

(ix) $\mathrm{wlp}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}}(x)$ *is the largest solution* y *of* $(\mathrm{d}_B + x) \cdot (\overline{\mathrm{d}_B} + \mathrm{wlp}_S(y)) = y$,

(x) $\mathrm{wp}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}}(x)$ *is the smallest solution* y *of* $(\mathrm{d}_B + x) \cdot (\overline{\mathrm{d}_B} + \mathrm{wp}_S(y)) = y$,

(xi) $\mathrm{wlp}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}}(x) = \sum\{y : (\mathrm{d}_B + x) \cdot (\overline{\mathrm{d}_B} + \mathrm{wlp}_S(y)) = y\}$,

(xii) $\mathrm{wp}_{\mathsf{do}\, B\rightarrow S\, \mathsf{od}}(x) = \prod\{y : (\mathrm{d}_B + x) \cdot (\overline{\mathrm{d}_B} + \mathrm{wp}_S(y)) = y\}$.

Theorems 14 and 8 show that wlp.(-) and wp.(-) qualify as predicate transformer semantics according to the requirements of [9]. The requirement [9, R0, p. 132] (which also appears as [9, (0), p. 129]), that $\mathrm{wlp}_S(\text{-})$ distribute over arbitrary meets, holds by Theorem 8(ix). Note that correctness of the interpretation is not needed for R0. Definitions [9, (10)–(18), pp. 133–136], which specify $\mathrm{wlp}_S(\text{-})$ and $\mathrm{wp}_S(\text{-})$ in case S is havoc, abort, or skip, hold by Theorem 14(i)(ii)(iii). Definitions [9, (23)–(25), p. 137], which specify the predicate transformers for the composition of commands $S_0;S_1$, hold by Theorem 14(iv)(v). Definitions [9, (27)–(29), p. 137], for the alternative construct if i:B_i→S_i fi, hold by Theorem 14(vi)(vii). Finally, Definitions [9, (1)–(2), p. 171], for the repetitive construct do B→S od, hold by Theorem 14(ix)(x).

The equation $\mathrm{r}_S;1 + \mathrm{e}_S = 1$ asserts that every state initiates either a terminating or a nonterminating computation of S [9, p. 130]. This equation is equivalent to $\mathrm{wp}^*_S(1) = 1$ and equivalent to $\mathrm{wp}_S(0) = 0$. This last equation has been called the "law of the excluded miracle". Theorem 15 below shows that the basic commands havoc, abort, and skip satisfy this "law" under any correct interpretation, and that if the other basic commands also do so then *all* commands do so and the interpretation is "miracle-free", *i.e.*, $\mathrm{wp}_S(0) = 0$ for every command S. Any miracle-free interpretation gives rise to predicate transformers that satisfy all the requirements of [9].

Theorem 15. (i) $\mathrm{wp}_{\mathrm{havoc}}(0) = \mathrm{wp}_{\mathrm{abort}}(0) = \mathrm{wp}_{\mathrm{skip}}(0) = 0$.

(ii) *If* $\mathrm{wp}_{S_0}(0) = 0$ *and* $\mathrm{wp}_{S_1}(0) = 0$ *then* $\mathrm{wp}_{S_0;S_1}(0) = 0$.

(iii) *If* $\mathrm{wp}_{S_i}(0) = 0$ *for every* $i \in I$, *then* $\mathrm{wp}_{\mathrm{if}\, i:B_i \to S_i\, \mathrm{fi}}(0) = 0$.

(iv) *If* $\mathrm{wp}_S(0) = 0$ *then* $\mathrm{wp}_{\mathrm{do}\, B \to S\, \mathrm{od}}(0) = 0$.

(v) *If* $\mathrm{wp}_S(0) = 0$ *for every basic command* S, *then* $\mathrm{wp}_S(0) = 0$ *for every command* S.

From their operational interpretation it is natural to expect that skip and abort should be deterministic. It is also natural to say that havoc is not deterministic, since, in the operational interpretation, a computation of havoc can start at any machine state and end at any other. However, even under the operational interpretation there is one case in which havoc really *is* deterministic, namely, when there is only one machine state. These ideas are expressed formally in the following theorem.

Theorem 16. (i) skip *and* abort *are deterministic.*

(ii) havoc *is deterministic if and only if* $\mathfrak{A}$ *is Boolean, i.e.*, $1' = 1$.

Some obviously sufficient (but not necessary) conditions for determinism are given next.

Theorem 17. (i) *If* S_0 *and* S_1 *are deterministic, then so is* $S_0;S_1$.

(ii) *Assume* S_i *is deterministic for every* $i \in I$ *and* $\mathrm{d}_{B_i} \cdot \mathrm{d}_{B_j} = 0$ *whenever* $i \neq j$ *and* $i, j \in I$. *Then* if $i{:}B_i \to S_i$ fi *is deterministic.*

(iii) *If* S *is deterministic, then so is* do $B \to S$ od.

Next is a generalization of what is called "the Main Repetition Theorem" for do $B \to S$ od in [9]. An informal statement of this result runs as follows. Assume

(i) P is a predicate,

(ii) if P and B hold at some state p then no nonterminating computation of S is possible from p,

(iii) if P and B hold at the initial state p_1 of a terminating computation of S, then P holds at the final state p_2, and the initial state p_1 is in the relation G to (is "greater than") the final state p_2, *i.e.*, $\langle p_1, p_2 \rangle \in G$,

(iv) there is no infinite sequence of states such that P and B hold at every state in the sequence, and each state is in relation G to the next state.

It follows from these assumptions that $\mathrm{wp}_{\mathrm{do}\, B \to S\, \mathrm{od}}(P)$ holds where P does, that is, P is a sufficient (but usually not necessary) condition for the guaranteed termination of do $B \to S$ od at a state satisfying P. Theorem 18 generalizes the Main Repetition Theorem in two ways. First, it does not include the assumption that G is transitive, a possibility noted in [9, pp. 174–5]. Second, it applies to interpretations over arbitrary complete relation algebras, not just representable relation algebras of the form $\mathfrak{Re}(U)$.

Theorem 18. *Assume* $\mathfrak{A}, \mathrm{r}, \mathrm{e}, \mathrm{d}$ *is a correct interpretation of* $\mathcal{L}$, S *is a command, and* B *is a predicate. For all* p, g *in* $\mathfrak{A}$, *if*

(i) $p;1 = p$,

(ii) $p \cdot \mathrm{d}_B \cdot \mathrm{e}_S = 0$,

(iii) $p \cdot \mathrm{d}_B \cdot \mathrm{r}_S \leq g \cdot \breve{p}$,

(iv) $\sum\{z : z \leq p \cdot \mathrm{d}_B \cdot g;(p \cdot \mathrm{d}_B \cdot z)\} = 0$,

then $p \leq \mathrm{wp}_{\mathrm{do}\, B \to S\, \mathrm{od}}(p)$.

References

1. James F. Allen, *An interval-based representation of temporal knowledge*, Proceedings of the Seventh International Joint Conference on Artificial Intelligence, (IJCAI), 1981, pp. 221–226.
2. ———, *Maintaining knowledge about temporal intervals*, Communications of the Association for Computing Machinery 26(11) (November 1983), 832–842.
3. James F. Allen and Patrick J. Hayes, *A commonsense theory of time*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1985, pp. 528–531.
4. ———, *Moments and points in an interval-based temporal logic*, Tech. Report TR 180, Department of Computer Science, University of Rochester, December 1987.
5. James F. Allen and Johannes A. Koomen, *Planning using a temporal world model*, Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, W. Germany, August 1983 (IJCAI), 1983, pp. 741–747.
6. George Boole, *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities*, Walton and Maberley, London, 1854.
7. ———, *The mathematical analysis of logic; being an essay towards a calculus of deductive reasoning*, B. Blackwell, Oxford, 1948, first published in London and Cambridge, 1847.
8. Louise H. Chin and Alfred Tarski, *Distributive and modular laws in the arithmetic of relation algebras*, University of California Publications in Mathematics, New Series 1 (1951), 341–384.
9. Edager W. Dijkstra and Carel S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, New York-Berlin-Heidelberg, 1990.
10. Michael R. Garey and David S. Johnson, *Computers and Intractibility, A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.
11. Leon Henkin, J. Donald Monk, and Alfred Tarski, *Cylindric Algebras, Part I*, North-Holland, Amsterdam, 1971.
12. ———, *Cylindric Algebras, Part II*, North-Holland, Amsterdam, 1985.
13. C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorenson, J. M. Spivey, and B. A. Sufrin, *Laws of programming*, Communications of the A. C. M. (August, September 1987), 672–686, 770.
14. C. A. R. Hoare and He Jifeng, *The weakest prespecification, Part I*, Fundamenta Informatica 9 (1986), 51–84.
15. ———, *The weakest prespecification, Part II*, Fundamenta Informatica 9 (1986), 217–252.
16. Bjarni Jónsson, *Varieties of relation algebras*, Algebra Universalis 15 (1982), 273–298.
17. ———, *The theory of binary relations*, Algebraic Logic (Proc. Conf. Budapest 1988) (Amsterdam) (H. Andréka, J. D. Monk, , and I. Németi, eds.), Colloq. Math. Soc. J. Bolyai, vol. 54, North-Holland, 1991, pp. 245–292.
18. Bjarni Jónsson and Alfred Tarski, *Boolean algebras with operators, Part II*, American Journal of Mathematics 74 (1952), 127–162.
19. Peter B. Ladkin and Roger D. Maddux, *Representation and reasoning with convex time intervals*, Tech. Report KES.U.88.2, Kestrel Institute, April 1988.
20. ———, *On binary constraint problems*, Tech. Report TR 102, Department of Computing Science and Mathematics, University of Stirling, April 1992, revised February 1993, to appear in the Journal of the Association for Computing Machinery.
21. ———, *On binary constraint networks*, Tech. Report KES.U.88.8, Kestrel Institute, November 1988.
22. Roger C. Lyndon, *The representation of relational algebras*, Annals of Mathematics (series 2) 51 (1950), 707–729.
23. Roger D. Maddux, *A working relational model: The derivation of the Dijkstra-Scholten predicate transformer semantics from Tarski's axioms for the Peirce-Schröder calculus of relations*, to appear in the South African Computer Journal.
24. ———, *The working relational model for predicate transformer semantics*, submitted to Theoretical Computer Science.
25. ———, *Topics in Relation Algebras*, Ph.D. thesis, University of California, Berkeley, 1978, pp. iii+241.
26. ———, *Some varieties containing relation algebras*, Transactions of the American Mathematical Society 272 (1982), 501–526.
27. ———, *Finite integral relation algebras*, Universal Algebra and Lattice Theory, Springer-Verlag, 1985, Proceedings of the Southeastern Conference in Universal Algebra and Lattice Theory, Charleston, S.C., July 11-14, 1984, Lecture Notes in Mathematics 1149, pp. 175–197.
28. ———, *Introductory course on relation algebras, finite-dimensional cylindric algebras, and their interconnections*, Algebraic Logic (Proc. Conf. Budapest 1988) (Amsterdam) (H. Andréka, J. D. Monk, and I. Németi, eds.), Colloq. Math. Soc. J. Bolyai, vol. 54, North-Holland, 1991, pp. 361–392.
29. ———, *Pair-dense relation algebras*, Transactions of the American Mathematical Society 328 (1991), 83–131.
30. ———, *The origin of relation algebras in the development and axiomatization of the calculus of relations*, Studia Logica 50 (3/4) (1991), 421–455.

31. J. Malik and T. O. Binford, *Reasoning in time and space*, Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, W. Germany, August 1983 (IJCAI), 1983, pp. 343–345.

32. J. M. Martin, *Dictionary of Philosophy and Psychology*, Macmillan & Co., New York, 1911, second edition.

33. J. Donald Monk, *On representable relation algebras*, Michigan Mathematical Journal 11 (1964), 207–210.

34. Augustus De Morgan, *On the symbols of logic, the theory of the syllogism, and in particular of the copula, and the application of the theory of probabilities to some questions in the theory of evidence*, Transactions of the Cambridge Philosophical Society 9 (1856), 79–127, reprinted in [36].

35. ———, *On the syllogism, no. IV, and on the logic of relations*, Transactions of the Cambridge Philosophical Society 10 (1864), 331–358, reprinted in [36].

36. ———, *On the Syllogism, and Other Logical Writings*, Yale University Press, New Haven, 1966, edited, with an Introduction by, Peter Heath.

37. Charles Sanders Peirce, *Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic*, Memoirs of the American Academy of Sciences 9 (1870), 317–378, reprinted by Welch, Bigelow and Co., Cambridge, Mass., 1870, pp. 1–62; also reprinted in [40] and [41].

38. ———, *On the algebra of logic*, American Journal of Mathematics 3 (1880), 15–57, reprinted in [40].

39. ———, *Note B: the logic of relatives*, Studies in Logic by Members of the Johns Hopkins University (Boston) (C. S. Peirce, ed.), Little, Brown, and Co., 1883, book reprinted, with an Introduction by Max H. Fisch and a Preface by Achim Eschbach, by John Benjamins Publishing Co., Amsterdam and Philadelphia, 1983, pp. lviii, vi+203; paper reprinted in [40], pp. 187–203.

40. ———, *Collected Papers, Volume III*. Harvard University Press, Cambridge, 1933, edited by Charles Hartshorne and Paul Weiss.

41. ———, *Writings of Charles S. Peirce, A Chronological Edition* , Indiana University Press, Bloomington, 1984, edited by Edward C. Moore, Max H. Fisch, Christian J. W. Kloesel, Don D. Roberts, and Lynn A. Ziegler.

42. F. W. K. Ernst Schröder, *Vorlesungen über die Algebra der Logik (exacte Logik), Volume 3, Algebra und Logik der Relative, part I*, second ed., Chelsea, Bronx, New York, 1966, first published in Leipzig, 1895.

43. R G. Simmons, *The use of quantitative and qualitative simulations*, Proceedings of Third National Conference on Artificial Intelligence (AAAI-83) Washington, D. C., August 1983, 1983.

44. Alfred Tarski, *On the calculus of relations*, The Journal of Symbolic Logic 6 (1941), 73–89.

45. ———, *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics 5 (1955), 285–309.

46. Alfred Tarski and Steven R. Givant, *A Formalization of Set Theory without Variables*, Colloquium Publications, vol. 41, American Mathematical Society, 1987.

47. P. G. van Beek, *Reasoning about qualitative temporal information*, Proceedings of AAAI-90, the Eighth National Conference on Artificial Intelligence, AAAI Press, 1990, pp. 728–734.

48. P. G. van Beek and R. Cohen, *Approximation algorithms for temporal reasoning*, Proceedings of IJCAI89, the 11th Joint Conference on Artifical Intelligence, Morgan Kaufmann, 1989, short version of [49], pp. 1291–1296.

49. ———, *Exact and approximate reasoning about temporal relations*, Computational Intelligence 6 (1990), 132–144, long version of [48].

50. J. F. A. K. van Benthem, *The Logic of Time*, Reidel, 1983.

51. M. Vilain and H. Kautz, *Constraint propagation algorithms for temporal reasoning*, Proceedings of AAAI-86, Morgan Kaufmann, 1986, pp. 377–382.

52. M. Vilain, H. Kautz, and P. G. van Beek, *Constraint propagation algorithms for temporal reasoning*, Readings in Qualititative Reasoning About Physical Systems (Weld and de Kleer, eds.), Morgan Kaufmann, 1989, revised version of [51].

53. Alfred North Whitehead and Bertrand Russell, *Principia Mathematica, Volume I*, Cambridge University Press, Cambridge, England, 1910, Second edition, 1925.

DEPARTMENT OF MATHEMATICS, 400 CARVER HALL, IOWA STATE UNIVERSITY, AMES, IOWA 50011-2066, U.S.A.

E-mail address: maddux@vincent.iastate.edu

Category theory and information system engineering

Michael Johnson and C.N.G. Dampney

School of Mathematics and Computing, Macquarie University
AUSTRALIA

Abstract

This paper is a summary of a talk for AMAST 1993. The actual talk contains examples drawn from business applications which because of confidentiality agreements cannot be published here. It is hoped that we will obtain permission to publish the examples in the final paper.

We outline a number of applications of category theory to information system engineering in major business enterprises. These applications have led to new methodologies in ER-modelling, constraint specification and process modelling. They also suggest new but as yet untested techniques for information system partitioning and architecture.

Our main thesis is that elementary category theoretic notions can have important value in the "real world" of software engineering.

1 Introduction

There have been many applications of category theory to computer science and these have been recorded in textbooks (eg [11] [1]) and conference proceedings (eg [3] [6]). Surprisingly few of these applications have yet filtered down to affect software engineering methodologies, and to the authors' knowledge none of them has influenced information system engineering methodologies (although there have been several category theoretic treatments of information systems, see [7] [10] [9] [5]). In this paper we record some elementary categorical observations about information systems and show how they have led to improved methodologies for information system engineering. The results reported here are essentially empirical, and are based on consultancy work that we have undertaken for Telecom Australia and Caltex Oil Australia as well as several smaller enterprises.

The paper is organised as follows. Section 2 briefly reviews information systems and the dominant methodology for planning information system designs which is ER-modelling. In Section 3 we review the definition of a category and indicate how an ER-model is essentially a category—the *classifying category* or *theory* for the information system. A brief analysis of this view shows that the categories that we need to deal with are at least lextensive [2] and that the category theoretic treatment gives a query language for free. In Section 4 we note that the main difference between an ER-model and its classifying category amounts to the specification of integrity constraints upon data which can be stored in the information system. This has led to a change in the main methodologies by giving constraint specification a much greater role in the development of information models. We show how to treat both static constraints and dynamic constraints (these latter are often business rules or government regulations which may be changed during the life of the information system). Section 5 treats process modelling which is traditionally the next stage in the development of a system after information (ER) modelling and it shows how the categorical treatment greatly simplifies process modelling. Finally Section 6 records implications of the category theoretic framework for developing different user views of an information system and for the underlying architecture of the system itself.

Overall our approach has become known as the Federated Information System (FIS) approach to information system engineering.

Acknowledgments: The authors gratefully acknowledge the Australian Research Council (ARC) and Caltex Oil Australia for supporting this research.

2 Information Systems

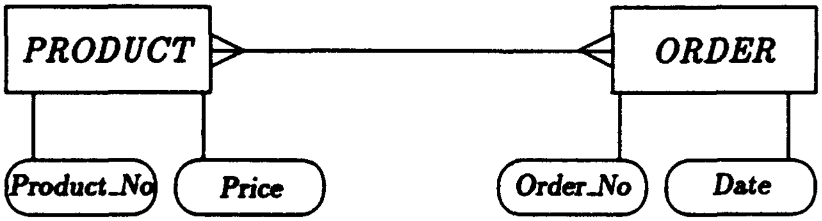
There is little need to discuss the importance and pervasiveness of computer technology in our society. Yet for those of us who work in academic institutions at least, it is easy to carry a biased view of the nature of its applications. Many of us focus on important issues such as algorithms and complexity and we often have a background in scientific computation. Yet the great majority of commercial applications require very little *computation*. Banks, airlines, stock exchanges, telephone utilities and even manufacturers and distributors use computers mostly to store, retrieve and perform simple transformations on information. The construction and maintenance of these *information systems* is the major expense item in many commercial information technology departments.

As is the case with most software engineering projects a great bulk of the expense in information system engineering occurs after the production version of the system has been produced because of the need for maintenance and modification. This expense can be substantially reduced if sufficient effort is expended in the planning stages to ensure that the information system is an accurate model of the business enterprise aspects which it is intended to support, and so information system analysts have concentrated on developing good methodologies for information system specification and development.

One of the easiest mistakes to make in developing an information system is to begin by considering what the organisation believes needs to be *done* by the system. As business develops these needs change rapidly, and modifying a system which has been designed to perform a particular task can be very difficult. Instead we should focus on what information the business needs to keep, and build a system which stores that information and which is able to utilise it as flexibly as possible. It is empirically well established that the underlying *information model* of a business changes relatively slowly and that the changes are usually incremental rather than revolutionary.

Thus, information system engineering usually begins with the development of an information model. There are several ways that such a model can be represented, but by far the dominant technique is called *Entity-Relationship* (ER) *modelling* [4].

The ER approach is a graphical modelling technique. An *entity* is a class of something about which the business needs to store information. Examples might include CUSTOMER, EMPLOYEE, ORDER, INVOICE and PRODUCT. Each entity will correspond to a set of things at a particular point in time (for example the current set of employees). The information that we store about entities comes in two forms: there are *relationships* between entities (for example an order may be for several products and a product may appear on several orders so there is a many-to-many relationship between PRODUCT and ORDER) and entities have certain *attributes* (for example a product may have a product number and a price, an employee has a name, an address, a salary and so on). Often one attribute for each entity is treated as the *key attribute* so that for example a product may be always accessed via its product number. Entities are usually represented graphically as rectangular boxes, relations as lines joining the boxes (with "crows-feet" to indicate possibly multi-valued relations when necessary), and attributes as oval boxes. An example is shown here.



The graphical nature of ER-models is a very important aspect of their popularity. Other specification techniques such as Z are more powerful but harder to learn. The great value of a graphical model is that an analyst can show it to businesspeople and with only a brief explanation they can understand and if necessary correct the model.

There is an extensive methodology of ER-modelling including the reduction of models to various normal forms. The details need not concern us here except for one aspect: Many-to-many relationships can always be transformed into two many-to-one relationships by the introduction of a new entity. For example we can introduce an entity ORDER_LINE. One instance of this entity will be the order of a particular product on a particular order. Thus there will be many-to-one relations (functions) between ORDER_LINE and PRODUCT and between ORDER_LINE and ORDER. (This is just the usual "tabulation" of a relation.)

3 Category theory

A category consists of a collection of *objects* and a collection of *arrows*, with each arrow having a specified source and target among the objects (this much is just a directed graph) together with a *composition* of arrows, defined whenever the arrows have a common source and target, which is associative and has identities.

Thus a category may be thought of as a directed graph together with information about composition. This information may be expressed by giving a set of relations (eg f composed with g is equal to h composed with k) and those relations are often expressed as *commutative diagrams* (a diagram is said to commute if any two composable paths of arrows in the diagram with common start point and common end point have equal composites).

Examples of categories include the category set whose objects are sets and whose arrows are functions between sets; grp whose objects are groups and whose arrows are group homomorphisms; more generally T-alg whose objects are algebras from some theory T and whose arrows are T-homomorphisms; and numerous *small categories* which can be generated by drawing a directed graph, adding identities at each vertex, and specifying composites for composable pairs of arrows.

One of the great advantages of category theory is that it has provided a graphical framework for much of mathematics. Many properties that seem to be about the internal structure of objects such as being a one element set or being the cartesian product of two sets, can be characterised by universal properties of arrows and these permit graphical arguments to prove theorems.

Further examples and definitions of specific universal properties such as pullback, terminal object and coproduct can be found in any of the basic texts [8] [1] [11].

3.1 A category theoretic view of an ER-model

We aim now to show how an ER-model is essentially a category. This is motivated by the categorical treatment of universal algebra above (the T-alg example) and is treated in full in [5]. By analogy with universal algebra we will call the category the *theory* or *classifying category* of the ER-model.

Consider an ER-model, normalised as described at the end of the preceding section so that all relations are many-to-one. This model may be viewed as a directed graph whose vertices are the entities and attributes of the model and whose arrows are the relationships oriented from the many-valued entity to the one-valued entity together with arrows from each entity to each of its attributes. Notice that if the vertices of this graph are thought of as sets, and the arrows as functions, then the intended semantics of the ER-model is still well represented here (and this can be made formal via a functor to set in the usual way).

It remains to consider composition. Since the many-to-one relations in the model are intended to represent real world many-to-one relations (functions) there are real world compositions and we argue that these should be represented in the model. Many of the compositions are free in the sense that formal composites can just be added to the model (or indeed left out since such formality can be added later), but when there is a closed loop of arrows it is important to determine, by considering the real world semantics, whether the diagram commutes. Once this has been done for all possible composites we have constructed a classifying category for the ER-model.

It is remarkable that extant ER-modelling methodologies have ignored this question of commuting diagrams. Typically an analyst spends a great deal of time and effort developing a model and eventually passes it to a programmer to implement. Often it is important that the resultant program check the constraint implied by the commutativity of certain diagrams, but since the analyst has not recorded which diagrams commute it is up to the less experienced programmer to try to reconstruct the intended semantics and to decide whether a given diagram should commute!

In fact, in our experience, searching for commutative diagrams actually results in a better ER-model because it often clarifies the nature of relationships and because it provides a test of the model as it is being developed. In the lecture this is illustrated by examples taken from commercial modelling exercises.

We view the specification of which diagrams commute in an ER-model as an important part of the information modelling methodology and we are developing CASE tools to assist in this process.

3.2 Classifying categories and lextensive categories

This section uses a little more category theory than the rest of this paper in order to accurately develop the notion of classifying category. It may be skipped by those with little category theoretic background who are mainly interested in our practical methodological results.

Universal algebra suggests a better version of the classifying category discussed above. Often an algebraic theory can be presented in several different ways, but there is a single canonical classifying category (up to equivalence of categories) obtained by taking any one of the presentations and "closing it up" under certain basic operations like taking limits. Similarly we would expect the classifying category of an information system to satisfy certain basic exactness properties and the category described in Section 3.1 is just a presentation for the canonical classifying category.

So, what basic exactness properties are required? We need a terminal object I and arrows $I \rightarrow A$ will be used to specify instances of the entity A. We need finite coproducts for two reasons. First, entities often have substructure which is best indicated by coproducts (so for example in a small retail business the entity EMPLOYEE might be the coproduct of the entities DRIVER, SALESPERSON, CLERICAL_STAFF and MANAGEMENT). Secondly, attributes are fixed sets (so for example PRODUCT_NO might be the set of all four digit numbers—of course most of these numbers won't be used at any particular point in time, but the relationship between PRODUCT and PRODUCT_NO allows us to see which ones are currently valid product numbers). Thus attributes are usually $\sum^n I$ for some n ($n = 10000$ in our product numbers example). This is technically very important since the injection $i_k : I \rightarrow \sum^n I$ allows us to pick out attribute number k from which, if the attribute is a key attribute, we can obtain information about a particular instance of the corresponding entity. Finally we need pullbacks, both to allow us to compose relations and to allow us to access the entity instances with particular attribute values.

Furthermore we expect the coproducts to behave well. They should be disjoint and universal. Thus in the presence of pullbacks and a terminal object we expect our classifying category to be a *lextensive category* [2].

3.3 The query language

For use in Section 4 it is worth noting that the internal logic of the lextensive classifying category of an information system forms a query language for that system. Thus the standard queries arise as objects of the classifying category.

Models of the information system will be lextensive functors from its classifying category to set. Such functors will necessarily carry the object representing a query to the set of records which satisfy the query.

4 Constraint specification

We show by example how to model the vast majority of the integrity constraints required in information systems by using ER-modelling with commutative diagrams. Some examples of the constraints which can be treated include the requirement that in a database of students, courses, classes and class times, it is required that no student have a clash between two timetabled classes; when an order is delivered it must be delivered to the address of the customer who placed the order; and when a contractor engages in some work involving a business resource there must be a contract that specifies that that contractor has the right to use that resource.

Some complicated constraints require the use of the query language outlined above (since a constraint may apply only to a certain subentities determined by a description that can be used as a query).

In the talk we show how both permanent (static) and variable (dynamic) constraints can be easily modelled.

5 Process modelling

Once a satisfactory ER-model has been developed it is common to work out a process model for the business which shows the important processes carried out by the business and how they trigger one another. The process model will be much less general than the ER-model since it will tell us about how the business is currently organised (and this may change).

Traditionally the process model is influenced by the ER-model, but our new methodology for ER-modelling makes the link explicit. Consider the diagrams in the ER-model which have been specified as commuting. Typically each of these loops represents an individual process and reconciliation cycle. This is because, in order to update the information system, it is usually necessary to update an instance at each vertex of the diagram and then finally to check that commutativity has been preserved.

Thus to develop the process model one calculates a kind of graph dual of the ER-model in which specified commutative diagrams correspond to processes and common vertices between such commutative diagrams correspond to triggers between the processes.

Of course it is often the case that an analyst can further refine the process model, but it is useful to note that the greater part of the work of developing a process model has already been done if one has specified the commutative diagrams in the ER-model.

Once again this point is illustrated with real world examples in the talk.

6 Views and architectural implications

The methodology that we have been describing also has some as yet untested implications for other aspects of information system development.

One particularly difficult problem in dealing with large information systems is the presentation of different *views* of the system for different users. The problem is essentially one of how to partition the system so that users can see a relatively complete view related to the aspects that are of relevance to them without having to look at the whole system. The recognition of commuting diagrams as processes suggests that the best partitioning would be obtained by choosing a related group of commutative diagrams. This will be developed in work currently in progress.

This partitioning can be carried further. The growth of very large information systems has led to problems of complexity and context retention which might best be solved by allowing business units a certain autonomy with their information systems. However, integration of such systems is necessary and the complexity of the interaction between subsystems can be dangerous. We propose the development of a corporate information (ER) model which can be used to determine, via commutative diagrams how to partition the system into subsystems. This will require duplicating entities that happen to fall into two subsystems and providing a message passing mechanism to allow the two copies to remain synchronised. However, if the partitioning is done well, and we believe an analysis based on commutative diagrams will do this, then it is likely that the interaction between subsystems will be quite manageable.

This proposed architecture for information systems is the source of the name Federated Information Systems.

References

[1] M. Barr and C. Wells, *Category theory and computer science*, Prentice Hall, 1990.

[2] A. Carboni, S. Lack and R.F.C. Walters, Introduction to extensive and distributive catgories, *Pure Mathematics Report* 92-9, University of Sydney, 1992.

[3] Category theory and computer science conferences, Springer lecture notes in computer science, **240**, **283** and **389**

[4] P.P.S. Chen, The entity relationship model—towards a unified view of data, *ACM Transactions on Database Systems*, 1 (1976), 9–36.

[5] C.N.G. Dampney, M. Johnson and G.P. Monro, An illustrated mathematical foundation for ERA, in C.M.I. Rattray and R.G. Clarke (eds) *The unified computation laboratory*, Institute of mathematics and its applications, (1992), 77–84.

[6] Durham Conference, Applications of categories in computer science, *London Mathematical Society Lecture Note Series* 177 (1992).

[7] C.A. Gunther, The mixed power domain, to appear in *Theoretical Computer Science*.

[8] Saunders Mac Lane, *Categories for the Working Mathematician*, Graduate Texts in Mathematics 5, Springer-Verlag, 1971.

[9] R. Rosebrugh and R.J. Wood, Relational databases and indexed categories, *Canadian Mathematical Society Conference Proceedings* 13 (1992), 391-407.

[10] S. Vickers, Geometric theories and databases, *London Mathematical Society Lecture Notes* 177 (1992), 288–314.

[11] R.F.C. Walters, *Categories and computer science*, Cambridge University Press, 1992.

Rigorous specification of real-time systems

Steve Schneider

May 1993

Abstract

This paper provides an introduction to the use of timed CSP in reasoning about real-time systems. The language of timed CSP and the denotational timed failures model are reviewed, and the underlying theory is discussed. The algebraic style of specification is discussed, followed by the behavioural specification approach. A simple timed buffer example is treated using both methods.

1 Introduction

A real-time system is one whose correct operation relies upon some consideration of its quantitative timed behaviour; examples include traffic lights, gas burners, washing machines, and nuclear power plants. Many specifications on such systems are concerned with explicit timing properties such as response time or delay time. To reason rigorously about them, it is necessary to be able to capture real-time properties in a precise way, and to have some model of computation that incorporates time.

There are a number of approaches that have been taken to provide a rigorous foundation for reasoning about real-time systems. One approach is to focus attention on specifications, providing a language suitable for capturing and reasoning about real-time requirements independent of any particular formalism for describing systems. Metric temporal logic [Koy89] and the duration calculus [ZHR91] are two examples. Such specification languages are generally supported mathematically by an underlying model, and may be used with a variety of system description formalisms.

The complementary approach is to begin with a way of describing processes. There are many ways timed systems may be described, including timed automata [AlD91], timed graphs [LyV91], timed petri nets (e.g. [CoR85]), a multitude of timed process algebras [BaB91, HeR91, MoT90, Che92, Wan91, ReR86, NiS90], timed versions of LOTOS [QaF87, BoL91]

Processes or abstract programs are often used as specifications in their own right, by treating them as descriptions of how a system is intended to behave.

In this case, another essential part of the specification is how a proposed implementation should relate to the specifying process. It may be required to be equivalent with respect to a set of axioms (as is often the case when the underlying semantics is axiomatic), or bisimilar, or testing equivalent (both with respect to an operational semantics), or equal in some denotational model. Alternatively, some notion of refinement may be preferred: a set of axioms might define a notion of refinement, or perhaps some simulation relation should hold between specification and implementation, or the implementation should pass more tests, or else their meanings should be related by some refinement in a denotational model.

In addition, the specification-oriented and the process-oriented approaches are often combined. A programming language may be provided together with an independent way of talking about properties. For example, timed graphs may be related to temporal logic [ACD93]; an occam-like language [Hoo91] may use metric temporal logic as a specification language, or a process algebra may be used in conjunction with a Hennessy-Milner style logic [HeM85]. Furthermore, any language with a denotational semantics will support specifications expressed directly as properties on subsets of the denotational model.

This paper describes two approaches that may be taken with the process algebra of timed Communicating Sequential Processes (CSP). It begins by reviewing the language, which is an extension of Hoare's CSP [Hoa85] which includes an explicit timing construct. Its denotational semantics is then given in terms of timed traces and timed refusals. The underlying theory is reviewed, as it is this theory which underpins all application of CSP. Finally, two approaches to specification are discussed. The language may be used as a specification language in the sense above, leading to processes as 'algebraic' specifications. The CSP approach to nondeterminism as underspecification leads naturally to a refinement relationship between a specification captured as a process, and a proposed implementation which should be at least as deterministic. The denotational semantics also makes it possible to capture requirements as properties on elements of the semantic model. This is done by specifying acceptable behaviour in a typical execution, and then requiring that all possible executions of a proposed implementation meet this specification.

2 Communicating Sequential Processes

In common with other process algebras, CSP is concerned purely with the communication patterns of processes, abstracting away internal state information which may be separated from communication behaviour. This abstraction remains appropriate for real-time systems since they are generally reactive, maintaining continual interaction with their environment. underlying model.

Events

A process is modelled in terms of the possible interactions it can have with its environment. These interactions are described in terms of instantaneous atomic synchronisations, or *events*. This notion of synchronisation is considered to be primitive: both asynchronous communication and communication by means of shared memory may be modelled in terms of it. When a process will be cooperating with its environment for some length of time, this is modelled in terms of an event at the point where they agree to cooperate. For example, a couple involved in a wedding service will be interacting for some time, yet there is a precise instant at which they become married.

Computational model

A number of assumptions are made about the underlying model of computation.

- **Maximal progress** A synchronisation event occurs as soon as all participants are ready to perform it.
- **Maximal parallelism** Every process has a dedicated processor; processes do not compete for processor time.
- **Finite variability** No process may perform infinitely many events, or undergo infinitely many state changes, in a finite interval of time.
- **Real-time** The time domain is taken to be the non-negative real numbers. Thus it is possible for events to occur at any non-negative real time. Since the reals are dense, our maximal parallelism assumption above means that there is no lower bound on the time difference between two independent events.

The language of CSP

Let Σ be the set of all possible events. The terms of CSP are given by the following Backus-Naur form:

$$\begin{array}{lll}
::= & \mathit{Stop} \mid \mathit{Skip} \mid P \,;\, P \mid a \longrightarrow P \mid & \text{sequential} \\
 & P \,\Box\, P \mid P \sqcap P \mid P \overset{t}{\triangleright} P \mid & \text{choice} \\
 & P \;{}_{A}\|_{A}\; P \mid P \,|||\, P \mid & \text{parallel} \\
 & P \setminus A \mid f(P) \mid f^{-1}(P) \mid & \text{abstraction} \\
 & X \mid \mu X \bullet P & \text{recursion}
\end{array}$$

where a is drawn from Σ, A is drawn from $\mathbf{P}(\Sigma)$, t from $[0, \infty)$, f is a function $\Sigma \rightarrow \Sigma$, and X is drawn from the set of process variables. CSP *processes* are terms with no free process variables (every process variable is bound by some μ expression), for which every recursive expression is guarded, as defined below.

The process *Stop* represents the deadlocked process, unable to engage in any events or make any progress. The process *Skip* is the immediately terminating process. A sequential composition $P \, ; Q$ initially behaves as P, but once P terminates, control is immediately passed, and the subsequent behaviour is that of Q. Thus we would expect $Skip \, ; P = P$ for any P, and $Stop \, ; P = Stop$, and indeed the semantic model supports these equations.

The prefix process $a \longrightarrow P$ is ready initially to engage in event a. It will continue to wait until its environment is also ready to perform it, at which point it will synchronise on this event. Once the event is performed its subsequent behaviour will be that of process P. There is no delay between the occurrence of a and the beginning of P.

An external choice $P \Box Q$ is initially ready to engage in events that either P or Q is ready to engage in. The first event performed resolves the choice in favour of the component that was able to perform it, and the subsequent behaviour is given by this component. If both components were able to perform the first event, then the choice is resolved nondeterministically.

An internal choice $P \sqcap Q$ behaves either as P or as Q, but unlike the external choice, the environment cannot influence the way the choice is resolved.

The timeout choice $P \overset{t}{\triangleright} Q$ initially behaves as process P. If an event is performed before time t, then the choice is resolved in favour of P which continues to execute, and Q is discarded. If no such event is performed, then the timeout occurs at time t, and the subsequent behaviour is that of Q.

The parallel combination $P \;{}_A\|_B\; Q$ allows P to engage in events from the set A, and Q to engage in events from the set B. The processes P and Q must synchronise on all events in the intersection $A \cap B$ of these two interfaces, but other events are performed independently. The asynchronous parallel combination $P \mid\mid\mid Q$ represents the independent concurrent execution of P and Q, with no synchronisation between them on any events.

The hiding operator $P \setminus A$ allows encapsulation of events in the set A; these events are made internal to the process, and are thus removed from the control of the environment. Since the cooperation of the environment is no longer required for these events, the only participants will be the components of P, and so the maximal progress assumption tells us that these internal events will occur as soon as P is ready to perform them. Hence internal events occur as soon as they are ready.

The interface renaming operators $f(P)$ and $f^{-1}(P)$ have the effect of changing the names of events through the alphabet mapping function f.

Recursion

A recursive term $\mu X \bullet P$ behaves as P, with every occurrence of X in P representing an immediate recursive invocation. Thus we will have the usual law

$$\mu X \bullet P = P[\mu X \bullet P / X]$$

We require that any recursive term of the form $\mu X \bullet P$ has that P is t-guarded for X for some $t > 0$. The following rules define when a timed CSP term P is t-guarded for variable X; a full discussion of t-guardedness can be found in [DaS93].

- For any X and t:
  1. $Stop$, $Skip$ are t-guarded for X
  2. X is 0-guarded for X
  3. $Y \neq X$ is t-guarded for X
  4. $\mu X \bullet P$ is t-guarded for X
- If P is t-guarded for X:
  1. $a \longrightarrow P$, $P \setminus A$, $f(P)$, $f^{-1}(P)$, and $\mu Y \bullet P$ are all t-guarded for X
- If P and Q are t-guarded for X:
  1. $P \Box Q$, $P \sqcap Q$, $P \,;\, Q$, $P ||| Q$, $P \;{}_A\|_B\; Q$ are all t-guarded for X
- If P is t-guarded for X, and Q is t'-guarded for X:
  1. $P \overset{u}{\rhd} Q$ is $\min\{t, u + t'\}$-guarded for X

Derived operators

A number of derived operators may be defined. The delay process $Wait\ t$, a timed form of $Skip$, which does nothing for t units of time and then terminates successfully, may be defined by the following equivalence:

$$Wait\ t \quad = \quad Stop \overset{t}{\rhd} Skip$$

The timeout choice waits for t units of time, but the process $Stop$ is unable to perform any event, and so the timeout will never be resolved in its favour. Thus at time t control is passed to $Skip$, which then terminates immediately.

A delayed form of prefixing may then be defined:

$$a \xrightarrow{t} P \quad = \quad a \longrightarrow (Wait\ t\,;P)$$

After the performance of event a, there is a delay of t units of time before control reaches P. The original version of timed CSP [ReR86, Ree88] treated prefixing as automatically delayed, with a constant delay δ. This would now be written as $a \xrightarrow{\delta} P$.

Generalising choice to allow infinite choices is often useful. The prefix choice $a : A \longrightarrow P_a$ is initially willing to perform any event from set A, and remains so willing until some event is chosen. Its subsequent behaviour, given by P_a, is dependent upon that event. Using this operator, an input construct can be defined, allowing the input on channel in of any item x in a set M:

$$in?x : M \longrightarrow Q(x) \quad = \quad a : in.M \longrightarrow P_a$$

where the set $in.M = \{in.m \mid m \in M\}$ and $P_{in.m} = Q(m)$ for every $m \in M$. The atomic synchronisation events here are of the form $in.m$.

Infinite nondeterministic choice may also be defined. The process $\bigsqcap_{j \in J} P_j$ for some indexing set J may behave as any of its arguments P_j. Thus for example a nondeterministic delay over some interval I may be defined:

$$Wait\ I \quad = \quad \bigsqcap_{t \in I} Wait\ t$$

This may delay for any time drawn from the interval I. If each of the P_i is t-guarded for X, then so is their infinite choice. Furthermore, if P is t-guarded for X, then $Wait\ I\,; P$ is $(t + \inf I)$-guarded for X

Finally, it is straightforward to generalise recursion to mutual recursion (finite or infinite); for further details see [DaS93].

A mathematical model

Notation

The variables t and u range over $\mathbf{R}^+$, the set of non-negative real numbers. Variable s ranges over $(\mathbf{R}^+ \times \Sigma)^*$, the finite sequences of timed events. We also use $\aleph \subseteq \mathbf{R}^+ \times \Sigma$.

We use the following operations on sequences: $\#s$ is the length of the sequence s; $s_1 \frown s_2$ denotes the concatenation of s_1 and s_2. We define the beginning and end of a sequence of timed events as follows: $begin(\langle(t, a)\rangle \frown s) = t$, $end(s \frown \langle(t, a)\rangle) = t$, and for convenience $begin(\langle\rangle) = \infty$ and $end(\langle\rangle) = 0$. The notation $s_1 \preceq s_2$ means that s_1 is a subsequence of s_2, and $s_1 \leq s_2$ means that s_1 is a prefix of s_2. The following projections on sequences are defined by list comprehension:

$$\begin{aligned}
s \lhd t &= \langle (u, a) \mid (u, a) \leftarrow s, u \leq t \rangle \\
s \mathrel{\lhd\!\!\mid} t &= \langle (u, a) \mid (u, a) \leftarrow s, u < t \rangle \\
s \rhd t &= \langle (u, a) \mid (u, a) \leftarrow s, u \geq t \rangle \\
s \uparrow I &= \langle (u, a) \mid (u, a) \leftarrow s, u \in I \rangle \\
s \upharpoonright A &= \langle (u, a) \mid (u, a) \leftarrow s, a \in A \rangle \\
s \setminus A &= \langle (u, a) \mid (u, a) \leftarrow s, a \notin A \rangle \\
s - t &= \langle (u - t, a) \mid (u, a) \leftarrow s, u \geq t \rangle
\end{aligned}$$

$$\begin{aligned} s + t &= \langle (u + t, a) \mid (u, a) \leftarrow s \rangle \\ strip(s) &= \langle a \mid (u, a) \leftarrow s \rangle \\ \sigma(s) &= \{a \mid s \upharpoonright \{a\} \neq \langle\rangle\} \end{aligned}$$

We also define a number of projections on sets of timed events:

$$\begin{aligned} \aleph \lhd t &= \{(u, a) \mid (u, a) \in \aleph, u < t\} \\ \aleph \rhd t &= \{(u, a) \mid (u, a) \in \aleph, u \geq t\} \\ \aleph \upharpoonright A &= \{(u, a) \mid (u, a) \in \aleph, a \in A\} \\ \aleph - t &= \{(u - t, a) \mid (u, a) \in \aleph, u \geq t\} \\ \sigma(\aleph) &= \{a \mid (u, a) \in \aleph\} \\ end(\aleph) &= sup\{u \mid (u, a) \in \aleph\} \end{aligned}$$

We will use $(s, \aleph, u) - t$ as an abbreviation for $(s - t, \aleph - t, max\{0, u - t\})$, and $end(s, \aleph)$ for $\max\{end(s), end(\aleph)\}$.

Observations

To provide a denotational semantics for the CSP operators, we construct a model of possible meanings for processes. This will be given in terms of observations that may be made of processes as they execute. We first define a set of possible observations *OBS*, and then associate with each process the set of observations that may be made of it. Processes are considered to be the same if the associated sets of observations are identical.

Any observation of an execution of a process must include a record of those events that were performed, and the times at which they occurred. A *timed trace* is a finite sequence of timed events, drawn from the set $[0, \infty) \times \Sigma$, such that the times associated with the events appear in non-decreasing order. Formally, we define the set *TT* of all possible timed traces as

$$TT \;=\; \{s \in ([0, \infty) \times \Sigma)^* \mid \langle (t_1, a_1), (t_2, a_2) \rangle \preceq s \Rightarrow t_1 \leq t_2\}$$

Timed traces provide much information concerning the possible executions of processes. But these systems are reactive, and so we are also interested in knowing when they will be able to interact with an environment which is ready to perform certain events, and when they will not be able to do so. Although this information may be deduced from the trace information in the case of deterministic systems, trace information is not sufficient in the case of nondeterministic systems. For example, the traces of

$$a \longrightarrow Stop \quad \text{and} \quad Stop \sqcap a \longrightarrow Stop$$

are the same, yet the first must always respond in an environment in which a is ready, whereas the second may not.

We will therefore also record *timed refusal* information. A timed refusal is made up of those events (with times) which the process refused to engage in during an execution. Our assumption of finite variability allows us to simplify the treatment of such sets. Since a process will continue to refuse an event while it remains in the same state, and since only finitely many state changes are possible in a finite time, we may consider a timed refusal as a step function from times to sets of events, containing the information about which set of events may be refused at which times.

Refusal information at a time t is considered to be subsequent to the events recorded in the trace at that time. For example, in the process

$$a \longrightarrow Stop \;\Box\; b \longrightarrow Stop$$

the event b cannot be refused before any events have occurred. But at the instant a occurs, the possibility of b is withdrawn and so it may be refused from t onwards. Thus we consider the step function to be closed at the lower end of a step, and open at the upper end. Observe that once a single a has occurred, then it too may be refused from that instant onwards, since no further copies of a are possible for the process.

Refusal sets are formally defined as those sets of events which can be expressed as finite unions of refusal tokens; this captures the required step structure:

$$\begin{array}{rcl} RTOK & = & \{[b,e) \times A \mid 0 \leq b < e < \infty \wedge A \subseteq \Sigma\} \\ RSET & = & \{\bigcup R \mid R \in \mathbb{F}(RTOK)\} \end{array}$$

A single observation will consist of a *timed failure*, made up of a trace $s \in TT$, a refusal set $\aleph \in RSET$, and a time $t < \infty$ which is the duration of the observation, and must therefore be greater than or equal to all times mentioned in s and $\aleph$. The trace and refusal are both recorded during the same execution. The information that $(s, \aleph, t)$ is an observation of P tells us that P has some execution up to time t during which the events in s were performed and the events in $\aleph$ were refused. In contrast to the untimed failures model for CSP, this refusal contains information concerning events that were refused before, during and after the performance of s, whereas an untimed refusal set contains only information after the end of the trace.

There is also another, contrapositive, view of the information contained in $(s, \aleph, t)$: as a partial record of an experiment on the process P. We may consider $\aleph$ as containing some information about what the environment of P was ready to perform, and the trace s may be considered as the response of P in this environment. For example, if we place the process $Wait\ 1\ ;\ a \longrightarrow Stop$ in an environment which is ready to perform a at all times between 0 and 2, then we would expect to see the event a occur at time 1. This corresponds to the observation $(\langle(1, a)\rangle, [0, 2) \times \{a\}, 2)$. This is also expected under the previous view: before time 1, the event a is not possible, and so it may be refused. At time 1, the event a occurs, and any further occurrences are not possible; since the refusal at time 1 is subsequent to the trace, we allow a to be refused at time 1, and from that time onwards. Thus the event a may be refused over the interval $[0, 2)$ provided one copy of it is performed at time 1. The information that a is also refused at time 1 simply states that the process may refuse to engage in further copies of that event.

Our set of observations is thus given by

$$OBS \;=\; \{(s, \aleph, t) \mid end(s) \leq t \wedge end(\aleph) \leq t\}$$

and processes will be associated with subsets of OBS.

The model

We identify a number of healthiness conditions, or axioms of the model TM_F which we would expect any set of observations consistent with some process to meet. Thus the timed failures model TM_F is defined to be those subsets S of OBS satisfying the following conditions:

1. $(\langle\rangle, \{\}, 0) \in S$
2. $(s \frown s', \aleph, t) \in S \Rightarrow (s, \aleph \lhd begin(s'), t) \in S$
3. $(s, \aleph, t) \in S \wedge \aleph' \in RSET \wedge \aleph' \subseteq \aleph \Rightarrow (s, \aleph', t) \in S$
4. $(s, \aleph, t) \in S \Rightarrow$
$$\begin{array}{ll} \exists \aleph' \in RSET \bullet & \aleph \subseteq \aleph' \wedge (s, \aleph', t) \in S \\ & \wedge\, \forall (u, a) \in [0, t) \times \Sigma \bullet \\ & \qquad (u, a) \notin \aleph' \Rightarrow (s \triangleleft u \frown \langle(u, a)\rangle, \aleph' \lhd u, u) \in S \\ & \qquad \wedge \\ & \qquad (u > 0 \wedge \neg \exists \varepsilon > 0 \bullet ((u - \varepsilon, u) \times \{a\} \subseteq \aleph')) \\ & \qquad\quad \Rightarrow (s \lhd u \frown \langle(u, a)\rangle, \aleph \lhd u, u) \in S \end{array}$$
5. $\forall n, \exists u \bullet \forall s, \aleph \bullet (s, \aleph, u) \in S \Rightarrow \#s < n$
6. $(s, \aleph, t) \in S \Rightarrow \forall u \geq end(s, \aleph) \bullet (s, \aleph, u) \in S$

Axiom 1 states that the empty observation must be possible. Axioms 2 and 3 state that if a particula: observation is possible, then observations containing less information must also be possible. Axiom 4 essentially states that any event must be possible or refusible: that there must be a 'complete' extension of the observed refusal set such that any timed event not in the complete refusal could have been performed. Axiom 5 enforces a speed limit on processes, which implies finite variability. Axiom 6 states that the same traces and refusals may be observed however long the process is watched. This last axiom implies that the duration information is redundant; the possible durations may be deduced from the trace and refusal information alone. However, we retain this information as an aid to specification.

The semantic function $\mathcal{F}_T$

We provide semantics only for the basic terms of the language without free variables; this may be lifted in the usual way [DaS93] to the language containing free variables.

The semantic function

$$\mathcal{F}_T : TCSP \longrightarrow TM_F$$

is defined by the following set of equations:

$$\mathcal{F}_T[\![Stop]\!] \ \widehat{=}\ \{(s, \aleph, t) \mid s = \langle\rangle\}$$

$$\begin{aligned}\mathcal{F}_T[\![Skip]\!] \ \widehat{=}\ & \{(\langle\rangle, \aleph, t) \mid \surd \notin \sigma(\aleph)\} \\ & \cup \\ & \{(\langle(u, \surd)\rangle, \aleph, t) \mid t \geq u \geq 0 \wedge \surd \notin \sigma(\aleph \lhd u)\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_T[\![P\,;Q]\!] \ \widehat{=}\ & \{(s, \aleph, t) \mid \surd \notin \sigma(s) \wedge \\ & \qquad\qquad (s, \aleph \cup ([0, t) \times \{\surd\})) \in \mathcal{F}_T[\![P]\!] \\ & \qquad \vee \\ & \qquad s = s_P \frown s_Q \wedge \surd \notin \sigma(s_P) \\ & \qquad \wedge (s_Q, \aleph, t) - u \in \mathcal{F}_T[\![Q]\!] \wedge begin(s_Q) \geq u \\ & \quad \wedge (s_P \frown \langle(u, \surd)\rangle, \aleph \lhd u \cup ([0, u) \times \{\surd\}), u) \in \mathcal{F}_T[\![P]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_T[\![a \longrightarrow P]\!] \ \widehat{=}\ & \{(\langle\rangle, \aleph, t) \mid a \notin \sigma(\aleph)\} \\ & \cup \\ & \{(\langle(u, a)\rangle \frown s, \aleph, t) \mid \ a \notin \sigma(\aleph \lhd u) \wedge \\ & \qquad\qquad\qquad (s, \aleph, t) - u \in \mathcal{F}_T[\![P]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_T[\![P \Box Q]\!] \ \widehat{=}\ & \{(\langle\rangle, \aleph, t) \mid (\langle\rangle, \aleph, t) \in \mathcal{F}_T[\![P]\!] \cap \mathcal{F}_T[\![Q]\!]\} \\ & \cup \\ & \{(s, \aleph, t) \mid s \neq \langle\rangle \wedge (s, \aleph, t) \in \mathcal{F}_T[\![P]\!] \cup \mathcal{F}_T[\![Q]\!] \\ & \qquad\qquad \wedge \\ & \qquad\qquad (\langle\rangle, \aleph \lhd begin(s), begin(s)) \in \mathcal{F}_T[\![P]\!] \cap \mathcal{F}_T[\![Q]\!]\}\end{aligned}$$

$$\mathcal{F}_T[\![P \sqcap Q]\!] \ \widehat{=}\ \mathcal{F}_T[\![P]\!] \cup \mathcal{F}_T[\![Q]\!]$$

$$\begin{aligned}\mathcal{F}_T[\![P \overset{u}{\rhd} Q]\!] \ \widehat{=}\ & \{(s, \aleph, t) \mid begin(s) \leq u \wedge (s, \aleph, t) \in \mathcal{F}_T[\![P]\!]\} \\ & \cup \\ & \{(s, \aleph, t) \mid begin(s) \geq u \wedge (\langle\rangle, \aleph \lhd u, u) \in \mathcal{F}_T[\![P]\!] \\ & \qquad\qquad \wedge \\ & \qquad\qquad (s, \aleph, t) - u \in \mathcal{F}_T[\![Q]\!]\}\end{aligned}$$

$$\mathcal{F}_T[\![P \ {}_A\|_B\ Q]\!] \ \widehat{=}\ \{(s, \aleph, t) \mid \exists \aleph_P, \aleph_Q \bullet$$

$$
\begin{aligned}
&\aleph \upharpoonright (A \cup B) = (\aleph_P \upharpoonright A) \cup (\aleph_Q \upharpoonright B)\\
&\wedge s = s \upharpoonright (A \cup B)\\
&\wedge (s \upharpoonright A, \aleph_P, t) \in \mathcal{F}_T\llbracket P\rrbracket\\
&\wedge (s \upharpoonright B, \aleph_Q, t) \in \mathcal{F}_T\llbracket Q\rrbracket \,\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_T\llbracket P \mathbin{|||} Q\rrbracket \;\widehat{=}\; \{(s, \aleph, t) \mid \exists\, s_P, s_Q \bullet\; & s \in s_P \mathbin{|||} s_Q \;\wedge\\
& (s_P, \aleph, t) \in \mathcal{F}_T\llbracket P\rrbracket \wedge\\
& (s_Q, \aleph, t) \in \mathcal{F}_T\llbracket Q\rrbracket\}
\end{aligned}
$$

where $s_P \mathbin{|||} s_Q$ is the set of timed traces consisting of an interleaving of s_P and s_Q.

$$
\begin{aligned}
\mathcal{F}_T\llbracket P \setminus A\rrbracket &\;\widehat{=}\; \{(s \setminus A, \aleph, t) \mid (s, \aleph \cup ([0, t) \times A), t) \in \mathcal{F}_T\llbracket P\rrbracket\}\\
\mathcal{F}_T\llbracket f(P)\rrbracket &\;\widehat{=}\; \{(f(s), \aleph, t) \mid (s, f^{-1}(\aleph), t) \in \mathcal{F}_T\llbracket P\rrbracket\}\\
\mathcal{F}_T\llbracket f^{-1}(P)\rrbracket &\;=\; \{(s, \aleph, t) \mid (f(s), f(\aleph), t) \in \mathcal{F}_T\llbracket P\rrbracket\}
\end{aligned}
$$

The infinite choice constructs are not always well defined, since axiom 5 might be violated if there is no speed limit which applies to all of the arguments simultaneously. We say that a set of processes R is *uniformly bounded* if the union $\bigcup R \subseteq OBS$ meets axiom 5. In such cases, the following definitions apply:

$$
\begin{aligned}
\mathcal{F}_T\llbracket \textstyle\bigsqcap_{i\in I} P_i\rrbracket &\;\widehat{=}\; \bigcup_{i\in I} \mathcal{F}_T\llbracket P_i\rrbracket\\
\mathcal{F}_T\llbracket a : A \longrightarrow P_a\rrbracket &\;=\; \{(\langle\rangle, \aleph, t) \mid A \cap \sigma(\aleph) = \{\}\}\\
&\qquad \cup \{(\langle(u, a)\rangle ^\frown s, \aleph, t) \mid\\
&\qquad\qquad a \in A \wedge A \cap \sigma(\aleph \lhd u) = \{\}\\
&\qquad\qquad \wedge (s, \aleph, u) - t \in \mathcal{F}_T\llbracket P(a)\rrbracket\}
\end{aligned}
$$

A full treatment of these operators requires a more complex model [MRS92, Sch92a].

In order to give a meaning to recursive constructs, the intention is that the recursive process $\mu X \bullet P$ should be a solution of the equation $X = P$. Thus we also allow recursive equations as process definitions: the equation $P = F(P)$ defines P to be the process $\mu X \bullet F(X)$.

It is by no means clear why such equations should have solutions at all, and we must impose some structure on the model in order to guarantee that they do. A distance function d between processes is defined:

$$
\begin{aligned}
S \lhd u &\;=\; \{(s, \aleph, t) \in S \mid t < u\}\\
d(S_1, S_2) &\;=\; \inf\{2^{-t} \mid S_1 \lhd t = S_2 \lhd t\}
\end{aligned}
$$

Thus the longer S_1 and S_2 are indistinguishable, the closer together they are under d. In fact, the distance function is a metric, and the space (TM_F, d) is a complete metric space [Ree88].

Now define a function $F(Y)$ to be t-constructive if

$$S_1 \lhd s = S_2 \lhd s \quad \Rightarrow \quad F(S_1) \lhd (s+t) = F(S_2) \lhd (s+t)$$

If a term P is t-guarded in X, it follows that the resulting function on X corresponds to a t-constructive function F on TM_F (for any instantiation of the other process variables). But this means that F is a contraction mapping: that is,

$$\exists \alpha < 1 \bullet \forall S_1, S_2 \bullet d(F(S_1), F(S_2)) \leq \alpha d(S_1, S_2)$$

where a suitable α is 2^{-t}. Thus we conclude from Banach's fixed point theorem [Sut75] that the function F has a *unique* fixed point in the complete metric space (TM_F, d).

It is now possible to give a meaning to a recursive term of the form $\mu X \bullet P$ for P t-guarded in X with $t > 0$. If P contains no free variables other than X then we have

$$\mathcal{F}_T[\![\mu X \bullet P]\!] \quad = \quad \text{The unique fixed point of the function corresponding to } \lambda X \bullet P$$

This approach may be lifted to terms P containing other free variables in the usual way, by evaluating the recursion while the values of the other variables remain fixed (see [DaS93]). It also extends easily to mutual recursion.

This semantic model corresponds in a natural way to an operational testing approach [Hen88] to identifying and distinguishing processes. An operational semantics of the language has been given [Sch92b] in terms of a timed transition system. A test is a CSP process T. A process P may pass a test T if there is some execution given by the operational semantics of $(P\ {}_{\Sigma}\|_{\Sigma}\ T) \setminus \Sigma$ for which T passes through a 'success' state. Two processes are equivalent under may testing if the set of tests they may pass are exactly the same. Then it turns out [Sch92b] that this notion of equivalence is the same as equivalence in the model TM_F: processes are equivalent under may testing precisely when they have exactly the same timed failures. Thus the denotational semantics is fully abstract with respect to the operational semantics. It follows that timed failures equivalence is the same as untimed and timed trace congruence.

3 Specification

3.1 Process algebra specification

As observed earlier, a common approach to specification is to use processes themselves as descriptions of required behaviour. By considering nondeterminism as underspecification, we consider an implementation or refinement of P to be a process Q which behaves as P but which may be more deterministic; some of the nondeterminism in P may be resolved in Q. Thus P is refined by Q if Q has fewer behaviours than P. This is written $P \sqsubseteq Q$, and defined

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \mathcal{F}_T[Q] \subseteq \mathcal{F}_T[P]$$

A process Q meets a specification P when $P \sqsubseteq Q$.

Consider for example a specification for calculating a square root of $|x|$:

$$SQRT \quad = \quad in?x \xrightarrow{1} ((out!(+\sqrt{|x|}) \longrightarrow Skip) \sqcap (out!(-\sqrt{|x|}) \longrightarrow Skip))$$

An internal choice is made as to whether to output the positive or the negative square root. If this specification process is suitable in a particular context, then the following refinement that is guaranteed to output the negative square root will also be suitable:

$$NSQRT \quad = \quad in?x \xrightarrow{1} out!(-\sqrt{|x|}) \longrightarrow Skip$$

The environment of the process cannot know whether the internal choice is resolved at run-time, or at compile-time, or by the implementor of the process.

The next specification is for a one-place buffer which takes between one second and eight seconds from inputting a message to enabling it for output. The next input may follow output immediately, and must be possible within five seconds of the last output.

$$B \quad = \quad in?x \longrightarrow Wait\,[1, 8]\,;\, out!x \longrightarrow Wait\,[0, 5)\,;\, B$$

Thus the process $Q = in?x \xrightarrow{3} out!x \xrightarrow{2} Q$ meets this specification, since it is a refinement of B; any possible behaviour of Q is also possible for B, and therefore acceptable.

As a larger example we will consider the following more complicated requirement: we wish to specify a process modelling an n-place unordered buffer of type T, which has certain constraints on input, output, and throughput:

- There must be at least 2 seconds between consecutive inputs;

- It must be ready to accept input no more than 10 seconds since the last input (if not full).

- There must be at least 4 seconds between consecutive outputs;

- It should always be ready to output within 10 seconds of the last output (if not empty).

- Any particular item must be available for output exactly 5 seconds after it is input, subject to the other constraints.

The first two constraints impose lower and upper bounds on the times at which the process should enable input. These are simultaneously captured by the following process:

$$IN \quad = \quad in?x : T \longrightarrow Wait\,[2, 10]\,; IN$$

Similarly, the bounds imposed by the next two constraints are captured by

$$OUT \quad = \quad out?x : T \longrightarrow Wait\,[4, 10]\,; OUT$$

Observe that OUT is prepared to allow any event of the form $out.m$; it is not constraining the nature of the output in any way, it is only constraining the time at which output becomes possible.

Finally, the fifth constraint may be captured for a buffer of size one as follows:

$$1BUFF \quad = \quad in?x : T \xrightarrow{5} out!x \longrightarrow 1BUFF$$

An unordered buffer of size n may be considered as a combination of n buffers of size 1 operating independently.

$$nBUFF \quad = \quad |||_{i=1}^{n}\, 1BUFF$$

where $|||_{i=1}^{n} P_i = P_1 \parallel P_2 \parallel \ldots \parallel P_n$. Since the interleaving operator $\parallel$ is associative (i.e. $\mathcal{F}_T[\![(P \parallel Q) \parallel R]\!] = \mathcal{F}_T[\![P \parallel (Q \parallel R)]\!]$), this is well defined.

The full specification may then be given as the parallel combination of these three specifications:

$$SPEC \quad = \quad (IN\ {}_{in.T}\|_{out.T}\ OUT)\ {}_{in.T \cup out.T}\|_{in.T \cup out.T}\ nBUFF$$

The event set associated with each component specification consists of those events that the specification is concerned with. The process IN imposes no constraints upon the events in $out.T$, so these events do not appear in its interface set, indicating that they can occur without the involvement of IN. Observe also that it is the constraint imposed by $nBUFF$ that prevents input when the buffer is full, and output when empty; the processes IN and OUT are not concerned with these aspects of the buffer's behaviour.

The compositional nature of the denotational semantics allows for a compositional treatment of refinement: if refinements of each of the specifications IN, OUT, and $nBUFF$ are independently found, then their parallel composition will be a refinement of the entire specification $SPEC$. This compositionality is essential for large-scale verification.

3.2 Behavioural specification

An alternative approach is to describe directly those observations that are acceptable, in terms of statements about traces and refusals. A specification in this style will be a predicate S on observations or behaviours, and a process P will meet a specification if the predicate holds for every observation in its semantics. In this case, we write P **sat** S, which is defined formally as follows:

$$P \ \mathbf{sat}\ S \quad = \quad \forall (s, \aleph, t) \in \mathcal{F}_T[P] \bullet S$$

This approach allows for a variety of levels of abstraction, since the specification S may be concerned only with some aspects of behaviour, and may ignore others. For example, an untimed safety specification associated with the square-root specification $SQRT$ above is that any answer given must be correct with respect to the input:

$$S1 \quad = \quad \forall x, y \bullet (\langle in.x, out.y\rangle) \leq strip(s) \Rightarrow y^2 = x$$

This specification has abstracted away any timing information, and is concerned purely with functional correctness. Timing properties are addressed by considering the times at which events occur.

$$S2 \quad = \quad \forall x, y, u_1, u_2 \bullet (\langle (u_1, in.x), (u_2, out.x)\rangle \preceq s) \Rightarrow u_1 + 1 \leq u_2$$

The specification $S2$ states that there must be a delay of at least one second between any input and any subsequent output.

All specifications that simply consider the trace s component of the observation are safety specifications, in Lamport's sense that 'nothing bad will happen': a constraint is imposed on which events are permissible and at what times. A process can fail such a specification only by performing some undesirable event. In particular, the deadlock process *Stop* will meet any satisfiable specification concerned simply with traces. A square root program could ensure it never gives the wrong answer simply by never giving any answer.

To specify that a process should make some progress, it is necessary to consider the refusal information. To say that the process is initially willing to accept any input, we require that it is unable to refuse input events to begin with:

$$S3 \quad = \quad s = \langle\rangle \Rightarrow in.T \cap \sigma(\aleph) = \{\}$$

To say that output must be available within one second of input we write

$$S4 = \forall u, x \bullet foot(s \upharpoonright in.T \cup out.T) = (u, in.x) \Rightarrow out.T \not\subseteq \sigma(\aleph \rhd (u + 1))$$

Recall that we also have an alternative view of refusals, as a partial record of what the environment of the process offered. The specification $S4$ may also be interpreted as saying that if the last event observed was some input at time u, then the environment cannot have been willing to accept output any time after $u+1$. This is equivalent to the previous reading because of the maximal progress property: if the environment *had* been willing, then all involved parties would have been ready and the output would have occurred. Read contrapositively, *S4* states that if the environment had offered to accept output, then something would have occurred after that last input (i.e. $(u, in.x)$ would not be the foot of the trace s).

This view of refusals also supports an assumption/commitment style of specification. It is often natural to specify what a process is expected to do, and then make explicit any assumptions about the environment. For example, the requirement that the three events a, b, and c are performed sequentially before time 1 is captured as follows:

$$C1 \;=\; t \geq 1 \Rightarrow \langle a, b, c \rangle \preceq strip(s \lhd 1)$$

If the observation lasts for at least one second, then the sequence $\langle a, b, c \rangle$ should appear in the trace by time 1.

No CSP process will be able to guarantee this specification unconditionally, since it could always be placed in an uncooperative environment which prevented these events from happening. But such a specification is generally made with the assumption that the events in question are under the control of the process required to perform them. This may correspond to an assumption that the environment is always willing to go along with the process with regard to these three events. This assumption is expressed as *A1*:

$$A1 \;=\; [0, t) \times \{a, b, c\} \subseteq \aleph$$

Then the resulting specification on a process is simply $A1 \Rightarrow C1$. This is met by a process such as $a \longrightarrow b \longrightarrow c \longrightarrow Stop$; observe that this process does not meet the specification *C1*, since $(\langle\rangle, \{\}, 1)$ is a possible observation of it for which *C1* fails.

As another example, consider the following specification on a buffer:

$$\begin{aligned} S5 \;=\; & [0, t) \times out.T \subseteq \aleph \Rightarrow \\ & \quad (\forall u, x \bullet (u, in.x) \in s \land u + 1 < t \Rightarrow (u + 1, out.x) \in s) \end{aligned}$$

Here the assumption about the environment is that it is always willing to accept output: for the duration t of the observation, all output events are present in $\aleph$, indicating that the environment was willing to accept all such events. Under this assumption, we require that for any time u at which a message x is input, a corresponding output must appear in the trace one second later (provided the observation lasts that long).

For the purposes of comparison and contrast, we return to the five requirements on the n-place unordered buffer. These are respectively rendered as behavioural specifications below. We must also make the n-place requirement explicit, in *B0*. Observe in *B1* and *B3* that the lower bound of the desired response time is captured by a trace specification stating that events cannot appear too close together; these are safety properties. The upper bound requirements given by *B2* and *B4* must be captured by an assertion about the readiness of the process to engage in further events by a particular time, expressed in terms of refusals. We do not insist that some event must be performed (unless we make an assumption about the environment), since a process does not have sole control over the performance of events.

$$\begin{array}{ll}
B0. & 0 \le \#(s \upharpoonright in.T) - \#(s \upharpoonright out.T) \le n \\
B1. & \forall u \bullet \#(s \upharpoonright in.T \uparrow (u, u+2)) \le 1 \\
B2. & (\#(s \upharpoonright in.T) - \#(s \upharpoonright out.T) < n) \Rightarrow \\
 & \qquad in.T \cap \sigma(\aleph \rhd end(s \upharpoonright in.T) + 10) = \{\} \\
B3. & \forall u \bullet \#(s \upharpoonright out.T \uparrow (u, u+4)) \le 1 \\
B4. & (\#(s \upharpoonright in.T) - \#(s \upharpoonright out.T) > 0) \Rightarrow \\
 & \qquad out.T \cap \sigma(\aleph \rhd end(s \upharpoonright out.T) + 10) = \{\} \\
B5. & ((s+5) \upharpoonright in.T)\ before_{in,out}\ (s \upharpoonright out.T)
\end{array}$$

where $s\ before_{in,out}\ s'$ holds if every output event $out.m$ in s' has some corresponding input event $in.m$ in s. It may be defined as follows:

$$s\ before_{in,out}\ s' \quad \Leftrightarrow \quad bag(s' \downarrow out) \subseteq bag(s \downarrow in)$$

where $(s \downarrow c)$ is the sequence of messages m that appear in s on channel c (i.e. when $c.m$ is in the trace); and $bag(s \downarrow c)$ is the corresponding bag of messages.

This last specification illustrates a feature of the model-based approach: that we always have the opportunity to provide new definitions appropriate for particular applications.

Thus we would expect our algebraic specification process $SPEC$ to meet the conjunction of these requirements:

$$SPEC \quad \mathbf{sat} \quad B0 \wedge B1 \wedge B2 \wedge B3 \wedge B4 \wedge B5$$

Verification

The composition nature of the denotational semantics allows for a specification oriented proof system for establishing claims of the form P **sat** S. A proof obligation on a compound process P can be reduced or factored into proof obligations on its components.

For example, the following rule is given for lockstep parallel composition:

$$\frac{\begin{array}{l} P_1\ \mathbf{sat}\ S_1 \\ P_2\ \mathbf{sat}\ S_2 \\ \forall \aleph \bullet [(\exists \aleph_1, \aleph_2 \bullet \aleph = \aleph_1 \cup \aleph_2 \wedge S_1[\aleph_1/\aleph] \wedge S_2[\aleph_2/\aleph]) \Rightarrow S] \end{array}}{P_1\ {}_{\Sigma}\|_{\Sigma}\ P_2\ \mathbf{sat}\ S}$$

Thus to prove that a parallel combination meets S, it is sufficient to find S_1 and S_2 which the components meet and whose combination implies S.

The proof system, containing a rule for each operator, is given in [DaS90]. The soundness of the rules follows from the semantic equations. The rules are also complete, in the sense that if the conclusion is true, then there are specifications S_1 and S_2 such that the antecedents are all simultaneously true.

The rule for recursion is also straightforward:

$$\frac{\begin{array}{l}\exists X \bullet X \ \mathbf{sat}\ S \\ \forall X \bullet (X \ \mathbf{sat}\ S \Rightarrow P \ \mathbf{sat}\ S)\end{array}}{(\mu X \bullet P)\ \mathbf{sat}\ S}$$

Its soundness follows from the fact that any predicate on processes of the form X **sat** S is closed in the metric space TM_F, for any specification S; and that any contraction which maps a non-empty closed set into itself has its unique fixed point in the closed set.

Current and future research

Although an operational and denotational semantics for timed CSP have been given and shown to be equivalent, there is not yet an equivalent axiomatic semantics. There are many laws for transforming process descriptions [Ree88], but these laws do not form a complete set. An approach similar to that taken in [Che92] appears promising, and may complete the trinity of complementary semantic approaches. This would give more backup to the algebraic specification style, since the claim $P \sqsubseteq Q$ might then be established by equational reasoning, as it is equivalent to the claim $P = P \sqcap Q$. Different specification styles might be appropriate for different parts of a development, and could be used in tandem since they are unified by the underlying model.

Specification macros [Dav93] to make behavioural specifications more palatable are under investigation. Specification clichés are captured at a higher level to make requirements easier to read and understand. For example, the specification $S4$ stating that output should be available one second after input is rendered in at $t \Rightarrow out$ from $t + 1$.

Machine assisted verification is another area of great interest, both in terms of support for proofs that processes meet behavioural specifications, and also in terms of the model-checking approach for algebraic specifications. The latter approach is based upon operational semantics, and the states of a proposed implementation are explored and checked against corresponding states in the specification.

Acknowledgements

This material draws on the work of many researchers involved with the timed CSP group in Oxford. In particular it is a pleasure to acknowledge the contributions made by Bill Roscoe, Mike Reed, Tony Hoare, Jim Davies, Dave Jackson, and Gavin Lowe.

I am also grateful to the UK Science and Engineering Research Council for their support under research fellowship B91/RFH/312.

References

[ACD93] R. Alur, C. Courcoubetis, and D. Dill, *Model-checking in dense real-time*, Information and Computation, 1993.

[AlD91] R. Alur and D. Dill, *The theory of timed automata*, Proceedings, Real-Time: Theory in Practice, LNCS 600, 1991.

[BaB91] J.C.M Baeten and J.A. Bergstra, *Real-time process algebra*, Formal Aspects of Computing, 1991.

[BoL91] T. Bolognesi and F. Lucidi, *Timed process algebras with urgent interactions and a unique powerful binary operator*, Proceedings, Real-Time: Theory in Practice, LNCS 600, 1991.

[Che92] Liang Chen, *Timed processes: models, axioms and decidability*, PhD thesis, University of Edinburgh, 1992.

[CoR85] J.E. Coolahan and N. Roussopoulos, *A timed Petri net methodology for specifying real-time system timing constraints*, in "Proceedings, International Workshop on Timed Petri Nets," Torino, Italy, 1985.

[DaS90] J.W. Davies and S.A. Schneider, *Factorising proofs in Timed CSP*, Proceedings of the Fifth Workshop on the Mathematical Foundations of Programming Language Semantics LNCS 442, 1990.

[DaS93] J.W. Davies and S.A. Schneider, *Recursion induction for real-time processes*, Formal Aspects of Computing, to appear, 1993.

[Dav93] J.W. Davies, *Specification and proof in real-time CSP*, Cambridge University Press, 1993.

[HeM85] M. Hennessy and A.J.R.G. Milner, *Algebraic laws for nondeterminism and concurrency*, Journal of ACM, 1985.

[Hen88] M. Hennessy, *An algebraic theory of processes*, M.I.T press, 1988.

[HeR91] M. Hennessy and T. Regan, *A process algebra for timed systems*, report 5/91, University of Sussex, 1991.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hoo91] J Hooman, *Specification and compositional verification of real-time systems*, PhD thesis, Eindhoven University of Technology, The Netherlands, 1991.

[Koy89] R.L.C. Koymans, *Specifying message passing and time-critical systems with temporal logic*, PhD thesis, Eindhoven University of Technology, The Netherlands, 1989.

[LyV91] N. Lynch and F. Vaandrager, *Forward and backward simulations for timing-based systems*, Proceedings, Real-Time: Theory in Practice, LNCS 600, 1991.

[MoT90] F. Moller and C. Tofts, *A temporal calculus of communicating systems*, Proceedings, CONCUR 1990, LNCS 458, Springer-Verlag.

[MRS92] M.W. Mislove, A.W. Roscoe, and S.A. Schneider, *Fixed points without completeness*, submitted for publication, 1992.

[NiS90] X. Nicollin, X and J. Sifakis, *The algebra of timed processes ATP: theory and application*, RT-C26, Projet SPECTRE, Laboratoire de Génie Informatique de Grenoble, 1990

[QaF87] J. Quemada and A. Fernandes, *Introduction of quantitative relative time into LOTOS*, in "Protocol Specification, Testing and Verification VII" North Holland, 1987.

[Ree88] G.M. Reed, *A Uniform Mathematical Theory for Real-Time Distributed Computing*, Oxford University DPhil thesis, 1988.

[ReR86] G.M. Reed and A. W. Roscoe, *A Timed Model for Communicating Sequential Processes*, Proceedings of ICALP'86, LNCS 226, 1986; Theoretical Computer Science 58, pp 249-261, 1988.

[Sch92a] S.A. Schneider, *Unbounded nondeterminism for real-time processes*, Oxford University Technical Report 13-92, *1992*.

[Sch92b] S.A. Schneider, *An operational semantics for timed CSP*, Information and Computation, to appear. Also Oxford University Technical Report 1-92, 1992.

[Sut75] W.A. Sutherland, *Introduction to metric and topological spaces*, Oxford University Press, 1975.

[Wan91] Wang Yi, *A calculus of real time systems*, Ph.D. Thesis, Chalmers University of Technology, Sweden, 1991.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn, *A calculus of durations*, Information Processing Letters 40,5, 1991.

Full Abstraction and Expressiveness in Structural Operational Semantics

(preliminary report)

R.J. van Glabbeek[*]
Computer Science Department, Stanford University
Stanford, CA 94305, USA.
rvg@cs.stanford.edu

This paper explores the connection between semantic equivalences for concrete sequential processes, represented by means of transition systems, and formats of transition system specifications using Plotkin's structural approach. For several equivalences in the linear time – branching time spectrum a format is given, as general as possible, such that this equivalence is a congruence for all operators specifiable in that format. And for several formats it is determined what is the coarsest congruence with respect to all operators in this format that is finer than partial or completed trace equivalence. Finally for some of the formats a small language specified in this format is provided such that any operator specifiable in that format can already be expressed in this language.

1 Preorders and equivalences on labelled transition systems

Definition 1 A *labelled transition system (LTS)* is a pair $(\mathbb{P}, \longrightarrow)$ with $\mathbb{P}$ a set (of *processes*) and $\longrightarrow \subseteq \mathbb{P} \times A \times \mathbb{P}$ for A a set (of *actions*).

Notation: Write $p \xrightarrow{a} q$ for $(p, a, q) \in \longrightarrow$ and $p \xrightarrow{a}$ for $\exists q \in \mathbb{P} : p \xrightarrow{a} q$.

The elements of $\mathbb{P}$ represent the processes we are interested in, and $p \xrightarrow{a} q$ means that process p can evolve into process q while performing the action a. By an action any activity is understood that is considered as a conceptual entity on a chosen level of abstraction. Different activities that are indistinguishable on the chosen level of abstraction are interpreted as occurrences of the same action $a \in A$. Actions may be instantaneous or durational and are not required to terminate, but in a finite time only finitely many actions can be carried out (i.e. only *discrete* systems are considered).

Below several semantic preorders and equivalences will be defined on processes represented by means of labelled transition systems. These preorders can be defined in terms of the *observations* that an experimentator could make during a session with a process.

Definition 2 The set $\mathbb{O}_A$ of *potential observations* over an action set A is defined inductively by:
$T \in \mathbb{O}_A$. The trivial observation, obtained by terminating the session.
$a\varphi \in \mathbb{O}_A$ if $\varphi \in \mathbb{O}_A$ and $a \in A$. The observation of an action a, followed by the observation φ.
$\tilde{X} \in \mathbb{O}_A$ for $X \subseteq A$. The investigated system cannot perform further actions from the set X.
$X \in \mathbb{O}_A$ for $X \subseteq A$. The investigated system can now perform any action from the set X.
$\bigwedge_{i \in I} \varphi_i \in \mathbb{O}_A$ if $\varphi_i \in \mathbb{O}_A$ for all $i \in I$. The systems admits each of the observations φ_i.
$\neg\varphi \in \mathbb{O}_A$ if $\varphi \in \mathbb{O}_A$. (It can be observed that) ψ cannot be observed.

[*] This work was supported by ONR under grant number N00014-92-J-1974.

Definition 3 Let $(\mathbb{P}, \rightarrow)$ be a LTS, labelled over A. The function $\mathcal{O}_A : \mathbb{P} \rightarrow \mathcal{P}(\mathbb{O}_A)$ of *observations* of a process is inductively defined by the clauses below.

$$\begin{array}{lll}
(T) & T & \in \mathcal{O}_A(p) \\
(a) & a\varphi & \in \mathcal{O}_A(p) \text{ if } p \xrightarrow{a} q \wedge \varphi \in \mathcal{O}_A(q) \\
(F) & \widetilde{X} & \in \mathcal{O}_A(p) \text{ if } p \not\xrightarrow{a} \text{ for } a \in X \\
(R) & X & \in \mathcal{O}_A(p) \text{ if } p \xrightarrow{a} \text{ for } a \in X \\
(\wedge) & \bigwedge_{i \in I} \varphi_i & \in \mathcal{O}_A(p) \text{ if } \varphi_i \in \mathcal{O}_A(p) \text{ for all } i \in I \\
(\neg) & \neg\varphi & \in \mathcal{O}_A(p) \text{ if } \varphi \notin \mathcal{O}_A(p)
\end{array}$$

As the structure of the set A of actions will play no rôle of significance in this paper, the corresponding index will from here on be omitted. Below several sublanguages of observations are defined.

$\mathbb{O}_T$ $\varphi ::= T \mid a\psi$ the *(partial) trace* observations

$\mathbb{O}_{CT}$ $\varphi ::= T \mid a\psi \mid \widetilde{A}$ the *completed trace* observations

$\mathbb{O}_F$ $\varphi ::= T \mid a\psi \mid \widetilde{X}$ the *failure* observations

$\mathbb{O}_R$ $\varphi ::= T \mid a\psi \mid \widetilde{X} \wedge Y$ the *readiness* observations

$\mathbb{O}_{FT}$ $\varphi ::= T \mid a\psi \mid \widetilde{X} \wedge \psi$ the *failure trace* observations

$\mathbb{O}_{RT}$ $\varphi ::= T \mid a\psi \mid \widetilde{X} \wedge \psi \mid X \wedge \psi$ the *ready trace* observations

$\mathbb{O}_S$ $\varphi ::= T \mid a\psi \mid \bigwedge_{i \in I} \varphi_i$ the *simulation* observations

$\mathbb{O}_{CS}$ $\varphi ::= T \mid a\psi \mid 0 \mid \bigwedge_{i \in I} \varphi_i$ the *completed simulation* observations

$\mathbb{O}_{FS}$ $\varphi ::= T \mid a\psi \mid \widetilde{X} \mid \bigwedge_{i \in I} \varphi_i$ the *failure simulation* observations

$\mathbb{O}_{RS}$ $\varphi ::= T \mid a\psi \mid \widetilde{X} \mid X \mid \bigwedge_{i \in I} \varphi_i$ the *ready simulation* observations

$\mathbb{O}_B$ $\varphi ::= T \mid a\psi \mid \bigwedge_{i \in I} \varphi_i \mid \neg\psi$ the *bisimulation* observations

$\mathbb{O}_{nA}$ $\varphi ::= T \mid a\psi \ (\psi \in \mathbb{O}_{mA} \text{ for some } m < n) \mid \bigwedge_{i \in I} \varphi_i \mid \neg\psi$ the *n-nested action* observations

$\mathbb{O}_{nT}$ $\varphi ::= T \mid a\psi \mid \bigwedge_{i \in I} \varphi_i \ (\varphi_i \in \mathbb{O}_{mT} \text{ for some } m < n) \mid \neg\psi$ the *n-nested trace* observations

$\mathbb{O}_{nS}$ $\varphi ::= T \mid a\psi \mid \bigwedge_{i \in I} \varphi_i \mid \neg\psi \ (\psi \in \mathbb{O}_{mS} \text{ for some } m < n)$ the *n-nested simulation* observations

For each of these notions N, $\mathcal{O}_N(p)$ is defined to be $\mathcal{O}(p) \cap \mathcal{P}(\mathbb{O}_N)$.

Definition 4 Two processes $p, q \in \mathbb{P}$ are *N-equivalent*, denoted $p =_N q$, if $\mathcal{O}_N(p) = \mathcal{O}_N(q)$.
p is *N-prequivalent* to q, denoted $p \sqsubseteq_N q$, if $\mathcal{O}_N(p) \subseteq \mathcal{O}_N(q)$.

In Van Glabbeek [2] the observations above and the corresponding equivalences are motivated by means of testing scenarios, phrased in terms of 'button pushing experiments' on generative and reactive machines. There it is also observed that restricted to the domain of finitely branching, concrete, sequential processes, most semantic equivalences found in the literature 'that can be defined uniformly in terms of action relations' coincide with one of the equivalences defined above. The same can be said for preorders. Here *concrete* refers to the absence of internal actions (τ-moves) or internal choice. In order to facilitate the connections with other work it is worth remarking that 2-nested trace equivalence is also known as *possible-futures* equivalence, and on the mentioned domain readiness equivalence coincides with *acceptance-refusal* equivalence, failure equivalence coincides with Hennessy and De Nicola's *(must) testing equivalence*, failure trace equivalence coincides with Phillips *refusal (testing)*, and ready trace equivalence coincides with *barbed* equivalence and with *exhibited behaviour* equivalence. In order to clarify a few more relations, the following *relational characterizations* of certain equivalences may be helpful.

Definition 5 Let $(\mathbb{P}, \longrightarrow)$ be an LTS. A *ready simulation* is a relation $R \subseteq \mathbb{P} \times \mathbb{P}$ satisfying

- $pRq \wedge p \xrightarrow{a} p' \ \Rightarrow \ \exists q' : q \xrightarrow{a} q' \wedge p'Rq'$
- $pRq \wedge p \not\xrightarrow{a} \ \Rightarrow \ q \not\xrightarrow{a}$

Theorem 1 $p \sqsubseteq_{RS} q$ iff $p \sqsubseteq_{FS} q$ iff there is a ready simulation R with pRq.

Proof: "$p \sqsubseteq_{RS} q \Rightarrow p \sqsubseteq_{FS} q$" is trivial. For "$p \sqsubseteq_{FS} q \Rightarrow$ there is a ready simulation R with pRq" it suffices to establish that $\sqsubseteq_{FS}$ is a ready simulation.

- Suppose $\mathcal{O}_{FS}(p) \subseteq \mathcal{O}_{FS}(q)$ and $p \xrightarrow{a} p'$. I have to show that $\exists q' \in \mathbb{P}$ with $q \xrightarrow{a} q'$ and $\mathcal{O}_{FS}(p') \subseteq \mathcal{O}_{FS}(q')$. Let Q be $\{q' \in \mathbb{P} \mid q \xrightarrow{a} q' \wedge \exists \varphi_{q'} \in \mathcal{O}_{FS}(p') - \mathcal{O}_{FS}(q')\}$. Then $a \bigwedge_{q' \in Q} \varphi_{q'} \in \mathcal{O}_{FS}(p) \subseteq \mathcal{O}_{FS}(q)$, so there must be a $q' \in \mathbb{P}$ with $q \xrightarrow{a} q'$ and $q' \notin Q$.
- Let $\mathcal{O}(p) \subseteq \mathcal{O}(q)$ and $p \stackrel{a}{\not\longrightarrow}$. Then $\widetilde{\{a\}} \in \mathcal{O}_{FS}(p) \subseteq \mathcal{O}_{FS}(q)$ and hence $q \stackrel{a}{\not\longrightarrow}$.

Finally I have to prove that for R a ready simulation one has $pRq \Rightarrow (\varphi \in \mathcal{O}_{RS}(p) \Rightarrow \varphi \in \mathcal{O}_{RS}(q))$. I will do so with induction on φ.

- Suppose pRq and $a\varphi \in \mathcal{O}_{RS}(p)$. Then there is a $p' \in \mathbb{P}$ with $p \xrightarrow{a} p'$ and $\varphi \in \mathcal{O}_{RS}(p')$. As R is a ready simulation, there must be a $q' \in \mathbb{P}$ with $q \xrightarrow{a} q'$ and $p'Rq'$. So by induction $\varphi \in \mathcal{O}_{RS}(q')$, and hence $a\varphi \in \mathcal{O}_{RS}(q)$.

The cases that φ is T, $\tilde{X}$, X or $\bigwedge_{i \in I} \varphi_i$ are straightforward. □

Definition 6 Let $(\mathbb{P}, \longrightarrow)$ be an LTS. A *simulation* is a relation $R \subseteq \mathbb{P} \times \mathbb{P}$ satisfying

- $pRq \wedge p \xrightarrow{a} p' \quad \Rightarrow \quad \exists q' : q \xrightarrow{a} q' \wedge p'Rq'$

A *bisimulation* is a symmetric simulation.

Theorem 2 $p \sqsubseteq_S q$ iff there is a simulation R with pRq.
$p \sqsubseteq_B q$ iff $p =_B q$ iff there is a bisimulation R with pRq.

2 Structural Operational Semantics

In this paper $\mathcal{V}$ and $\mathcal{N}$ are two disjoint countably infinite sets of *variables* and *names*. Many concepts that will appear are parameterized by the choice of $\mathcal{V}$ and $\mathcal{N}$, but as in this paper this choice is fixed, a corresponding index is suppressed.

Definition 7 *(Signatures).* A *function declaration* is a pair (f, n) of a *function symbol* $f \in \mathcal{N}$ and an *arity* $n \in \mathbb{N}$. A function declaration $(c, 0)$ is also called a *constant declaration.* A *signature* is a set of function declarations. The set $\mathbb{T}(\Sigma)$ of *terms* over a signature Σ is defined inductively by:

- $\mathcal{V} \subseteq \mathbb{T}(\Sigma)$,
- if $(f, n) \in \Sigma$ and $t_1, \ldots, t_n \in \mathbb{T}(\Sigma)$ then $f(t_1, \ldots, t_n) \in \mathbb{T}(\Sigma)$.

A term $c()$ is often abbreviated as c. For $t \in \mathbb{T}(\Sigma)$, $\mathcal{V}(t)$ denotes the set of variables that occur in t. $T(\Sigma)$ is the set of *closed* terms over Σ, i.e. the terms $t \in \mathbb{T}(\Sigma)$ with $\mathcal{V}(t) = \emptyset$. A Σ*-substitution* σ is a partial function from $\mathcal{V}$ to $\mathbb{T}(\Sigma)$. If σ is a substitution and S any syntactic object, then $S[\sigma]$ denotes the object obtained from S by replacing, for x in the domain of σ, every occurrence of x in S by $\sigma(x)$. In that case $S[\sigma]$ is called a *substitution instance* of S.

Definition 8 *(Transition system specifications).* Let Σ be a signature. A *positive* Σ*-literal* is an expression $t \xrightarrow{a} t'$ and a *negative* Σ*-literal* an expression $t \stackrel{a}{\not\longrightarrow}$ with $t, t' \in \mathbb{T}(\Sigma)$ and $a \in \mathcal{N}$. For $t, t' \in \mathbb{T}(\Sigma)$ the literals $t \xrightarrow{a} t'$ and $t \stackrel{a}{\not\longrightarrow}$ are said to *deny* each other. A *transition formula* over Σ is an expression of the form $\frac{H}{\alpha}$ with H a set of Σ-literals (the *antecedents* of the the rule) and α a Σ-literal (the *conclusion*). A formula $\frac{H}{\alpha}$ with $H = \emptyset$ is also written α. A literal or transition formula is *closed* if it contains no variables. An *action rule* is a transition formula with a positive conclusion. A *transition system specification (TSS)* is a pair (Σ, R) with Σ a signature and R a set of action rules over Σ. A TSS is *positive* if all literals in the antecedents of its rules are positive.

The concept of a TSS was introduced in GROOTE & VAANDRAGER [4]; the negative premisses were added in GROOTE [3]. The notion constitutes the first formalization of PLOTKIN's *Structural Operational Semantics (SOS)* [5] that is sufficiently general to cover most, if not all, of its applications.

Definition 9 *(Proof).* Let $P = (\Sigma, R)$ be a TSS. A *proof* of a transition formula $\frac{H}{\alpha}$ from P is a well-founded, upwardly branching tree of which the nodes are labelled by Σ-literals, such that:

- the root is labelled by α, and
- if β is the label of a node q and K is the set of labels of the nodes directly above q, then
  - either $K = \emptyset$ and $\beta \in H$,
  - or $\frac{K}{\beta}$ is a substitution instance of a rule from R,

If a proof of $\frac{H}{\alpha}$ from P exists, then $\frac{H}{\alpha}$ is *provable* from P, notation $P \vdash \frac{H}{\alpha}$.

Definition 10 *(Transition relation).* Let Σ be a signature. A *transition relation* over Σ is a relation $\longrightarrow \subseteq T(\Sigma) \times \mathcal{N} \times T(\Sigma)$. Elements (t, a, t') of a transition relation are written as $t \xrightarrow{a} t'$. Thus a transition relation over Σ can be regarded as a set of closed positive Σ-literals (*transitions*).

A positive TSS specifies a transition relation in a straightforward way as the set of all derivable transitions. But as pointed out in GROOTE [3], it is much less trivial to associate a transition relation to a TSS with negative premisses. Several solutions are proposed in [3] and [1]. The most general of those is through the notion of *stability*. It is not difficult to show that the concept of stability defined below is the same as that of Bol and Groote.

Definition 11 *(Stable transition relation).* Let $P = (\Sigma, R)$ be a TSS and let $\longrightarrow$ be a transition relation over Σ. $\longrightarrow$ is *stable* for P if:

$$\alpha \in \longrightarrow \;\Leftrightarrow\; \begin{array}{l}\text{there is a closed transition formula } \frac{H}{\alpha} \text{ without positive antecedents}\\ \text{with } P \vdash \frac{H}{\alpha} \text{ and } (t \xrightarrow{a} t') \in \longrightarrow \text{ for no } (t \stackrel{a}{\not\longrightarrow}) \in H \text{ and } t' \in T(\Sigma).\end{array}$$

According to BOL & GROOTE [1] the transition relation *associated* to a TSS is its unique stable transition relation if it exists. They argue that there is no satisfying way to accociate a tranition relation to a TSS that has no or multiple stable transition relations.

3 Formats and congruence theorems

Definition 12 *(ntyft/ntyxt-format).* An action rule $\frac{H}{t \xrightarrow{a} t'}$ over a signature Σ is in *ntyft-format* if t has the form $f(x_1, ..., x_n)$ for certain $(f, n) \in \Sigma$ and $x_1, ..., x_n \in V$, and all its positive antecedents have the form $t \xrightarrow{a} y$ with $y \in V - V(t)$. It is in *ntxft-format* if t has the form $x \in V$ and all its positive antecedents have the form $t \xrightarrow{a} y$ with $x \neq y \in V$. A TSS is in *ntyft/ntyxt-format* if all its rules are in *ntyft* or *ntyxt*-format.

Definition 13 The *bound* variables of an action rule $\frac{H}{t \xrightarrow{a} t'}$ over a signature Σ are inductively defined as the ones that occur in t or in the target s' of a positive antecedent $(s \xrightarrow{b} s') \in H$ where s contains bound variables only. The rule is *pure* if all variables that occur in it are bound, and a TSS is *pure* if it consists of pure rules only. A rule has *no lookahead* if all bound variables in the source of its antecedents also occur in the source of its conclusion. *Connectedness* is the smallest equivalence relation between the bound variables that appear in a rule such that x and y are connected if there is an antecedent $x \xrightarrow{a} y$.

Definition 14 A TSS is in *bisimulation format* if it is positive after reduction—as defined in [1]—and in *ntyft/ntyxt*-format. A TSS is in *nested simulation format* or *tyft/tyxt-format* if it is positive and in *ntyft/ntyxt*-format. A TSS is in *ready simulation format* if it is in bisimulation format and its rules have no lookahead. A TSS is in *ready trace format* if it is in ready simulation format and no two occurrences of variables in the target of a rule are connected in that rule. A TSS is in *failure format* if it is positive and in ready trace format.

Theorem 3 *(Congruence).* Bisimulation equivalence is a congruence for any TSS in bisimulation format. Similarly, n-nested simulation equivalence (for any $n \in \mathbb{N}$) is a congruence for any TSS in nested simulation format, Ready simulation equivalence is a congruence for any TSS in ready simulation simulation format, ready trace equivalence is a congruence for any TSS in ready trace format and failure equivalence as well as trace equivalence are congruences for any TSS in failure format.

4 Full abstraction

Definition 15 An equivalence is said to be *fully abstract* with respect to a set of operators L and another equivalence $\sim_{obs}$ if it is the coarsest congruence with respect to the operators in L that is finer that $\sim_{obs}$. An equivalence on labelled transition systems is *fully abstract* with respect to a TSS-format and an equivalence $\sim_{obs}$ if it is the coarsest congruence with respect to all operators specifiable by a TSS in that format that is finer that $\sim_{obs}$.

Theorem 4 Bisimulation equivalence is fully abstract w.r.t. the bisimulation format and trace equivalence. 2-nested simulation equivalence is fully abstract for the n-nested simulation format and completed trace equivalence. Simulation equivalence (=1-nested simulation equivalence) is fully abstract for the n-nested simulation format and trace equivalence. Ready simulation equivalence is fully abstract for the ready simulation simulation format and trace equivalence, as well as for the positive ready simulation format and completed trace equivalence. Ready trace equivalence is fully abstract for the ready trace format and trace equivalence. And failure equivalence is fully abstract for the failure format and completed trace equivalence.

5 Expressiveness

Robert de Simone has shown that any operator that can be specified in the failure format can be expressed in Meije (or any equivalent process algebraic language). I show that similarly any operator that can be specified in the positive ready simulation format can be expressed in a similar language to which an operator ! has been added. A similar result (with yet another operator) will be conjectured for the n-nested simulation format.

References

[1] R.N. Bol & J.F. Groote (1991): *The meaning of negative premises in transition system specifications (extended abstract).* In J. Leach Albert, B. Monien & M. Rodríguez, editors: *Proceedings* 18th *ICALP,* Madrid, *Lecture Notes in Computer Science* 510, Springer-Verlag, pp. 481–494. Full version appeared as Report CS-R9054, CWI, Amsterdam, 1990.

[2] R.J. van Glabbeek (1990): *The linear time – branching time spectrum.* In J.C.M. Baeten & J.W. Klop, editors: *Proceedings CONCUR 90,* Amsterdam, *Lecture Notes in Computer Science* 458, Springer-Verlag, pp. 278–297.

[3] J.F. Groote (1989): *Transition system specifications with negative premises.* Report CS-R8950, CWI, Amsterdam. An extended abstract appeared in J.C.M. Baeten and J.W. Klop, editors: *Proceedings CONCUR 90,* Amsterdam, LNCS 458, Springer-Verlag, 1990, pp. 332–341.

[4] J.F. Groote & F.W. Vaandrager (October 1992): *Structured operational semantics and bisimulation as a congruence. Information and Computation* 100(2), pp. 202–260.

[5] G.D. Plotkin (1981): *A structural approach to operational semantics.* Report DAIMI FN-19, Computer Science Department, Aarhus University.

Synchronous observers and the verification of reactive systems

Nicolas Halbwachs
Verimag Laboratory[*] and
Stanford University[†]

Fabienne Lagnier, Pascal Raymond
Verimag Laboratory[*]
B.P. 53X, 38041 Grenoble Cedex, France

Invited paper at the Third International Conference on Algebraic Methodology and Software Technology, AMAST'93, Twente, June 1993.

Introduction

Synchronous programming [IEE91, Hal93b] is a useful approach to design reactive systems. A synchronous program is supposed to *instantly* and *deterministically* react to events coming from its environment. The advantages of this approach have been pointed out elsewhere. Synchronous languages are simple and clean, they have been given simple and precise formal semantics, they allow especially elegant programming style. They conciliate concurrency (at least at the description level) with determinism. They can be compiled into a very efficient sequential code, by means of a specific compiling technique: The control structure of the object code is a finite automaton which is synthesized by an exhaustive simulation of a finite abstraction of the program.

Concerning program verification, it has been argued [BS91, HLR92a, Pnu92] that the practical goal, for reactive programs, is generally to verify some simple logical safety properties: By a *safety* property, we mean, as usual, a property which expresses that something will never happen, and by a *simple logical* property, we mean a property which depends on logical dependences between events, rather than on complex relations between numerical values.

For the verification of such properties also, the synchronous approach has some advantages: Since the parallel composition is synchronous, the desired properties of a program can be easily and modularly expressed by means of an *observer*, i.e., another program which observes the behavior of the first one and decides whether it is correct. Thus, the same language is used to write the program and its desired properties. The verification then consists in checking that the parallel composition of the program and its observer never causes the observer to complain. This verification can often be performed by traversing the finite control automaton built by the compiler. This automaton is generally much smaller than in the asynchronous case, where non-deterministic interleaving of processes is likely to result in state explosion.

An observer can also be used to express known properties of the program environment. As a reactive system is embedded into an environment with which it tightly interacts, the environment

[*] Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and Verilog SA associated with IMAG.

[†] This work was performed while the first author was on leave in Stanford University, partially supported by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-91-J-1901, and by a grant from the Stanford Office of Technology Licensing. This publication does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement of this work should be inferred.

must be strongly taken into account in program design and verification. Generally, the critical properties of a reactive system are only required to hold provided the environment also behaves correctly, that is, under some assumptions about the environment. In [HLR92b], we verified a very simple railways control system, and the most important part was the description of the realistic behavior of the trains (they obey the signals, they do not jump from one track to another, etc.). In [HLR92a], we used this ability of taking the environment into account in the verification, to propose a modular verification technique: When two processes run in parallel, each of them is part of the other's environment; so any property which is proved about one of them, can be used as an assumption about the other's environment.

So, our verification approach can be summarized by three simple ideas: (1) restriction to safety properties; (2) expression of these properties by means of a synchronous, deterministic observer; (3) taking into account assumptions about the environment. This paper is a survey of our specification and verification techniques, in a very general, language independent, framework. Section 1 introduces a simple model of synchronous input/output machines, which will be used throughout the paper. In section 2, we show how such a machine can be designed to check the satisfaction of a safety property, and we discuss the use of such an observer in program verification. In section 3, we use an observer to restrict the behavior of a machine. This is the basic way for representing assumptions about the environment. Applications to modular and inductive verification are considered. In modular verification, one has to find, by intuition, a property of a subprogram that be strong enough to allow the verification of the whole program without fully considering the subprogram. In section 4, we consider the automatic synthesis of such a property, and in section 5, we investigate the possibility of deducing the subprogram from such a synthesized specification.

1 Synchronous I/O machines

We first define an abstract model of synchronous reactive machines. As far as verification is concerned, we could use a synchronous process algebra [Mil81, Mil83, AB84] as a basic formalism. However, in the synthesis problem, we have to distinguish between inputs and outputs, since a process controls its outputs but not its inputs. So, we prefer to use a notion of synchronous machine where inputs and outputs do no play a symmetric role. In the following model, as in synchronous languages, outputs are non blocking and synchronously broadcast. Moreover, we will need an explicit notion of state, which lacks in process algebras.

1.1 Definitions

Let us consider a set S of *signals*, and let $E_S = 2^S$ be the set of *events*[1] on S. An *I/O machine* M is a 5-tuple $(Q_M, q0_M, I_M, O_M, \delta_M)$ such that

- Q_M is a set of states containing $q0_M$, the initial state;
- $I_M \subset S, O_M \subset S$ are the disjoint sets of input and output signals, respectively.
- $\delta_M \subseteq Q_M \times E_{I_M} \times E_{O_M} \times Q_M$ is the transition relation. When there is no ambiguity about the considered relation, we will often note "$q \xrightarrow[o]{i} q'$" instead of "$(q, i, o, q') \in \delta_M$".

[1] Events, with the union operation, will play the role of the *monoid of actions* in synchronous process algebras.

Intuitively, in response to a sequence $(i_1, i_2, \ldots, i_n, \ldots)$ of input events, such a machine returns a sequence $(o_1, o_2, \ldots, o_n, \ldots)$ of output events, such that there exists a sequence $(q_0, q_1, \ldots, q_n, \ldots)$ of states, with $q_0 = q0_M$ and for all $n \geq 1$, $q_{n-1} \xrightarrow[o_n]{i_n} q_n$. The sequence $((i_1 \cup o_1), (i_2 \cup o_2), \ldots, (i_n \cup o_n), \ldots)$ will then be called a *trace* of the machine.

If $\tau = ((i_1 \cup o_1), (i_2 \cup o_2), \ldots, (i_n \cup o_n))$ is a finite trace, and $(q_0, q_1, \ldots, q_n)$ is a corresponding sequence of states, we will note $q0_M \xrightarrow{\tau} q_n$. For any state q, we will note *traces*(q) the set $\{\tau \mid q0_M \xrightarrow{\tau} q\}$ of traces leading to q. This notation is extended to sets of states: For any $X \subseteq Q_M$, *traces*$(X) = \bigcup_{q \in X}$ *traces*(q).

Let us note δ^r_M the *reaction function* from $Q_M \times E_{I_M}$ into $2^{E_{O_M} \times Q_M}$, defined by

$$\delta^r_M = \lambda(q, i).\{(o, q') \mid (q, i, o, q') \in \delta_M\}$$

A *reactive* machine cannot refuse a non-empty input event, and thus satisfies the following property: $\forall q \in Q_M, \forall i \subseteq I_M,\ i \neq \emptyset \implies \delta^r_M(q, i) \neq \emptyset$.

A *deterministic* machine has at most one possible reaction to a given input event, and thus satisfies: $\forall q \in Q_M, \forall i \subseteq I_M,\ |\delta^r_M(q, i)| \leq 1$. For a deterministic machine, we will note δ^O_M (respectively δ^Q_M) the *function* giving, for a state q and an input event i, the output event o (resp. the next state q') such that (q, i, o, q') belongs to δ_M.

We will use the usual *precondition* and *postcondition* functions, from 2^{Q_M} to 2^{Q_M}: For any $X \subseteq Q_M$,

- *post*$_M(X)$ is the set of successors of states belonging to X:

$$post_M(X) = \{q' \mid \exists q \in X, \exists i, o,\ q \xrightarrow[o]{i} q'\}$$

- *pre*$_M(X)$ is the set of states having a successor state in X:

$$pre_M(X) = \{q \mid \exists q' \in X, \exists i, o,\ q \xrightarrow[o]{i} q'\}$$

- $\widetilde{pre}_M(X)$ is the set of states having *all* their successors in X:

$$\begin{aligned} \widetilde{pre}_M(X) &= \{q \mid \forall i, \forall o, \forall q' \text{ such that } q \xrightarrow[o]{i} q',\ q' \in X\} \\ &= Q_M \setminus pre_M(Q_M \setminus X) \end{aligned}$$

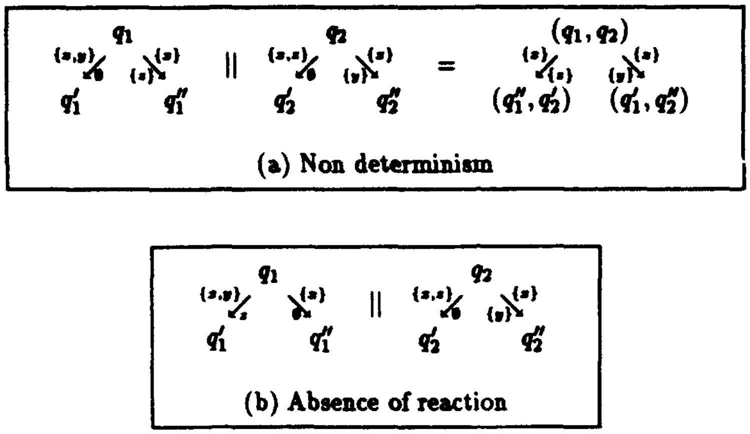
1.2 Operations on I/O machines

Projection: Let M be an I/O machine, and $O' \subseteq O_M$. The *projected machine* $M \downarrow O'$ is $(Q_M, q0_M, I_M, O', \delta')$, where $\delta' = \{(q, i, o \cap O', q') \mid (q, i, o, q') \in \delta_M\}$.

Obviously, if M is reactive (respectively, deterministic), so is $M \downarrow O'$.

Synchronous product: Let M_1 and M_2 be two I/O machines, with $O_{M_1} \cap O_{M_2} = \emptyset$ [2]. We define their *synchronous product* $M_1 || M_2$ to be the I/O machine M where

[2] The restriction that parallel machines don't share common output signals is for simplicity only. It does not exist in Esterel [BG92] and Argos [Mar92].



(a) Non determinism

(b) Absence of reaction

Figure 1: Synchronous product

- $Q_M = Q_{M_1} \times Q_{M_2}$, $q0_M = (q0_{M_1}, q0_{M_2})$
- $I_M = (I_{M_1} \setminus O_{M_2}) \cup (I_{M_2} \setminus O_{M_1})$, $O_M = O_{M_1} \cup O_{M_2}$
- $((q_1, q_2), i, o, (q_1', q_2')) \in \delta_M \iff (q_1, (i \cup o) \cap I_{M_1}, o \cap O_{M_1}, q_1') \in \delta_{M_1}$ and $(q_2, (i \cup o) \cap I_{M_2}, o \cap O_{M_2}, q_2') \in \delta_{M_2}$

In other words, a transition of the product involves a transition of each machine, triggered by the global input signals *and the signals emitted by the other machine.*

1.3 Causality

With this very loose definition of the synchronous product, it can happen that the product of two deterministic (respectively reactive) machines is not deterministic (resp. reactive). This is the well-known problem of *causality paradoxes* in synchronous languages [BG92, Mar92]. For instance, let $I_{M_1} = \{x, y\}, I_{M_2} = \{x, z\}, O_{M_1} = \{z\}$ and $O_{M_2} = \{y\}$. Then:

- Assume that $q_1 \xrightarrow[\emptyset]{\{x,y\}} q_1'$ and $q_1 \xrightarrow[\{z\}]{\{x\}} q_1''$ are the only transitions in δ_{M_1} from state q_1, and that $q_2 \xrightarrow[\emptyset]{\{x,z\}} q_2'$ and $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$ are the only transitions in δ_{M_2} from state q_2 (see Fig.1.a). If the input event $\{x\}$ occurs when the product machine $M_1 || M_2$ is in the state (q_1, q_2), two different transitions can take place:
  - either M_1 performs $q_1 \xrightarrow[\{z\}]{\{x\}} q_1''$ and then the emission of z forces the transition $q_2 \xrightarrow[\emptyset]{\{x,z\}} q_2'$ in M_2. So the compound transition is $(q_1, q_2) \xrightarrow[\{z\}]{\{x\}} (q_1'', q_2')$;
  - or, conversely, M_2 performs $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$, forcing the transition $q_1 \xrightarrow[\emptyset]{\{x,y\}} q_1'$ in M_1, and the resulting global transition is $(q_1, q_2) \xrightarrow[\{y\}]{\{x\}} (q_1', q_2'')$.

So, in that case, the product of two deterministic machines is non deterministic.

- Assume now that $q_1 \xrightarrow[\{z\}]{\{x,y\}} q_1'$ and $q_1 \xrightarrow[\emptyset]{\{x\}} q_1''$ are the only transitions in δ_{M_1} from state q_1, and that δ_{M_2} is as before (Fig. 1.b). Now, if the input event $\{x\}$ occurs in the state (q_1, q_2), no global transition can occur, since:
  - if M_2 performs $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$, then the emission of y forces the transition $q_1 \xrightarrow[\{z\}]{\{x,y\}} q_1'$ in M_1. But now, since z is emitted, M_2 should not have made its transition.
  - Conversely if M_1 performs $q_1 \xrightarrow[\emptyset]{\{x\}} q_1''$, since z is not emitted, M_2 must perform $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$ and the emission of y forbids the transition of M_1.

  So, in that case, the product of two reactive machines is not reactive.

An important feature of synchronous languages is that their parallel composition operator (synchronous product) introduces neither non-determinism nor deadlock. Compile-time consistency checks insure that the compound machine has a *unique, minimal*, reaction to each input event: Let M_1 and M_2 be two deterministic and reactive I/O machines, let $\delta^O_{M_1}$, $\delta^O_{M_2}$, be their respective output functions. When $M_1 || M_2$ is in the state (q_1, q_2) and receives an input event i, the output event o must satisfy

$$o = \delta^O_{M_1}(q_1, (i \cup o) \cap I_{M_1}) \cup \delta^O_{M_2}(q_2, (i \cup o) \cap I_{M_2})$$

i.e., be a fixpoint of the function $\lambda x.\, \delta^O_{M_1}(q_1, (i \cup x) \cap I_{M_1}) \cup \delta^O_{M_2}(q_2, (i \cup x) \cap I_{M_2})$. Causality problems come from the fact that this function is not always monotone, and thus, may admit zero or several minimal fixpoints. Compile-time consistency checks insure the existence and unicity of a least fixpoint, and the synchronous product is defined by

$$\delta^O((q_1, q_2), i) = \mu x.\, \delta^O_{M_1}(q_1, (i \cup x) \cap I_{M_1}) \cup \delta^O_{M_2}(q_2, (i \cup x) \cap I_{M_2})$$

$$\delta^Q((q_1, q_2), i) = (\delta^Q_{M_1}(q_1, (i \cup \delta^O((q_1, q_2), i)) \cap I_{M_1}), \delta^Q_{M_2}(q_2, (i \cup \delta^O((q_1, q_2), i)) \cap I_{M_2}))$$

(where, as usual, $\mu x.f$ denotes the least fixpoint of the function $\lambda x.f$).

2 Observers of safety properties

In this section, we show how a safety property can be specified by means of a synchronous observer. Such an observer is an I/O machine, taking as inputs both the input and the output signals of the machine under observation, and emitting an "alarm" signal as soon as the observed signals do not satisfy the property.

2.1 Safety properties

A *trace* τ on a set of signals S is a finite or infinite sequence of events on S. A *property* on S is a set of traces on S. An I/O machine M satisfies a property P if and only if each trace of M belongs to P. A property P on S is a *safety property* if and only if:

$\tau \in P \iff \tau' \in P$ for any finite prefix τ' of τ

In other words, a safety property is a prefix-closed (as expressed by the "$\Longrightarrow$" implication above) and limit-closed (as expressed by the "$\Longleftarrow$" implication) language on the vocabulary 2^S.

2.2 Observer

Let P be a safety property on S. Let α (read "alarm") be a signal not in S. An *observer* of P is a *deterministic* and *reactive* I/O machine $\Omega_P = (Q_{\Omega_P}, q0_{\Omega_P}, S, \{\alpha\}, \delta_{\Omega_P})$, returning a sequence of empty output events as long as it receives a sequence of input events which belongs to P. More precisely, let τ be a finite trace on S belonging to P (notice that the empty trace belongs to any safety property). Let q_τ be the state that Ω_P reaches after reading τ (if τ is the empty trace, q_τ is the initial state of Ω_P). Then, for any event $e \in 2^S$,

$$\delta^O_{\Omega_P}(q_\tau, e) = \begin{cases} \emptyset & \text{if } \tau.e \in P \\ \{\alpha\} & \text{otherwise} \end{cases}$$

Let us assume also that any transition emitting α leads to a distinguished state q_α.

Now, a machine M satisfies a safety property P if and only if the compound machine $M||\Omega_P$ never returns any event containing α; or, equivalently, never reaches an *erroneous* state belonging to $Q_M \times \{q_\alpha\}$. We will note Q_P^M the set $Q_M \times (Q_{\Omega_P} \backslash \{q_\alpha\})$ of non erroneous states of $M||\Omega_P$.

A practical advantage of this approach, is that the properties are written in the same language as the programs, and in fact, properties are programs. As such, they can be executed and tested. An observer can be actually run with the program, thus detecting any violation of the property (run-time checks).

Notice that this approach cannot be used with only an asynchronous composition, or at least, that it cannot be applied *modularly*. For instance, consider the following property: "*the signal* b *is emitted at least once between every two successive emissions of the signal* a ". If this property is checked by an asynchronous observer, since the observer is not guaranteed to catch all the signals, it can miss any occurrence of b. So, even if the property is satisfied, the observer can emit an alarm. To check such a property of an asynchronous program, one must add some synchronization code all along the transitions of the observed program, since otherwise, the asynchronous product does not ensure that all the transitions will be observed. When verifying a program, such modifications are of course harmful, since one cannot be sure that the verified program behaves the same once the additional code is removed. This contradicts G. Berry's "WYPIWYE" principle ("*what you prove is what you execute*") which fully applies in the synchronous case.

2.3 Application to program verification

The verification that a machine M satisfies a safety property P now amounts to proving that the machine $M' = M||\Omega_P$ never returns any event containing α. So, any safety property has been translated into an *invariant*. More precisely, one has to prove that the set *Reach*(M') of M' *reachable states* is included in the set Q_P^M of non erroneous states of M'. *Reach*(M') is classically defined as a least fixpoint:

$$Reach(M') = \mu X.\{q0_{M'}\} \cup post_{M'}(X)$$

Let us list the advantages of this expression of the verification problem, according to various verification methods:

State enumeration: For finite state systems, state enumeration techniques (enumerative model-checking) have been widely experimented [QS82, CES86]. In general, these techniques involve the construction of the whole state graph of the program, and its memorization for the analysis of *trace properties*. Now, since the problem has been reduced to the analysis of a *state property* (an invariant), the state graph needs only to be *traversed*. Particularly efficient techniques are available (e.g., [Hol87]) for such a traversal.

Reduction techniques: The drawback of state enumeration techniques is the explosion of the number of states, as the size of the program increases[3]. Other approaches [BRdSV90] consist in building a reduced state graph, according to some observation criteria. Now, in our approach, the machine of interest is not really $M||\Omega_P$, but rather $(M||\Omega_P) \downarrow \alpha$, since we are only interested in the presence of the signal α. This is an obvious observation criterion. So, in contrast with classic model-checking, the property is taken into account in the state graph generation. Assume the property is satisfied, then the minimal state graph of $(M||\Omega_P) \downarrow \alpha$ has only one state (it is the "always silent" automaton). Algorithms for generating a minimal state graph have been proposed [BFH+92, LY92]. When applied to our simple verification problem, these algorithms amount to proving that the initial state belongs to the greatest invariant $\mathit{Invar}(Q_P^M)$ included in Q_P^M, i.e., the greatest part of Q_P^M from which the transition relation does not permit to go out. This greatest invariant is wellknown to be a greatest fixpoint:

$$\mathit{Invar}(Q_P^M) = \nu X . Q_P^M \cap \widetilde{\mathit{pre}}_{M||\Omega_P}(X)$$

Approximate analysis: When infinite state systems are considered, approximate methods (and, in particular, *abstract interpretation techniques* [CC77, CC92]) can be applied to compute approximations of the set $\mathit{Reach}((M||\Omega_P) \downarrow \alpha)$. If an upper approximation of this set is included in Q_P^M, this proves that the erroneous states cannot be reached (see [Hal93a] for an application of such a method). If a lower approximation intersects the complement of Q_P^M, an error is detected.

In the remainder of the paper, we will essentially consider finite state machines, so all the considered fixpoints will be (theoretically) computable. In the following section, we will see that property observers can also be used to take into account known properties of the program environment.

3 Taking the environment into account

The main feature of reactive systems is that they tightly interact with their environment. As a consequence, the properties of the environment must be carefully taken into account in the design and verification of such a system. A reactive system is not intended to work in an *arbitrary* environment. In general, system specifications contain a lot of informations about the behavior of the environment, which are the hypotheses under which the design must take place. These known properties about the environment can involve not only the inputs of the system, but also its outputs, since the environment responds to the system. So, in general, among the set of traces of an I/O machine, only some of them are "realistic", i.e., satisfy the assumptions about the environment. In this section, we show how the behavior of an I/O machine can be restricted by a safety property, in order to take such assumptions into account in the verification process.

[3] Notice that the state explosion is more important in an asynchronous system, because of the non deterministic interleaving of asynchronous transitions.

3.1 Behavior restriction

Given a safety property A (assumption) of the environment of M, our goal is to define a *restricted* machine M' having exactly the same behaviors as M composed with any environment satisfying A: the set of traces of M' must be the intersection with A of the set of traces of M.

Restriction: Let M be an I/O machine, and Ω_A be an observer of a safety property A on the set $S = I_M \cup O_M$ of input/output signals of M. Let $M' = M||\Omega_A$. We define the restriction M/Ω_A to be the I/O machine $(Q_{M'}, q0_{M'}, I_M, O_M, \delta')$, where $\delta' = \{(q,i,o,q') \in \delta_{M'} \mid \alpha \notin o\}$

Obviously, the restricted machine M/Ω_A is generally not reactive, even if M is reactive: The restriction takes into account a property of the environment, and thus, refuses some unrealistic inputs. However, it can happen that in some states of the restricted machine, *all the input events* are refused. So, the restricted machine deadlocks, a highly undesirable situation in reactive systems. One can consider this as an error in the expression of the assumption A. However, we adopt another point of view: When restricting a machine M with an assumption A, the user intends to consider all the *infinite traces* of M that satisfy A. So, the machine must not enter any path in M/Ω_A which *inevitably* leads to a deadlock state. We define now another restriction, called *non-blocking restriction*, which has the intended behavior:

Non-blocking restriction: Let M be an I/O machine, and Ω_A be an observer of a safety property A on the set $S = I_M \cup O_M$ of input/output signals of M. Let $M' = M||\Omega_A$. Let us call $sink_A$ the set of states of M' leading inevitably to the violation of A:

$$sink_A = \mu X.\widetilde{pre}_{M'}((Q_M \times \{q_\alpha\}) \cup X)$$

Then, if $q0_{M'} \notin sink_A$, we define $M\underline{/}_{\infty}\Omega_A$ to be the I/O machine $(Q_{M'} \setminus sink_A, q0_{M'}, I_M, O_M, \delta'')$, where

$$\begin{aligned} \delta'' &= \delta_{M'} \cap ((Q_{M'} \setminus sink_A) \times E_{I_M} \times E_{O_M} \times (Q_{M'} \setminus sink_A)) \\ &= \{(q,i,o,q') \in \delta_{M'} \mid q, q' \notin sink_A \text{ and } \alpha \notin o\} \end{aligned}$$

One can notice that, if M is deterministic, $M\underline{/}_{\infty}\Omega_A = M/\Omega_{traces(Q_{M'} \setminus sink_A)}$. So, the property A has been strengthened into the other property $A' = traces(Q_{M'} \setminus sink_A)$ which cannot block the machine M: Any finite trace satisfying A' leads to state of M which has at least one outgoing transition preserving A'.

3.2 Application

As before, a direct use of this way of expressing assumptions by an observer, is to execute the observer with the program, thus checking at run-time that the assumptions are satisfied. The restriction can also be used for program testing, to use only testcases corresponding to realistic scenarios. We consider now the use of restriction in the verification process:

Verification under assumptions: Given an I/O machine M, a safety assumption A about its environment, and a safety property P, one can prove that M satisfies P provided the environment satisfies A, by

1. proving that $(M\underline{/}_{\infty}\Omega_A)$ has some behaviors, i.e., that the initial state of $M||\Omega_A$ does not belong to $sink_A$. Otherwise, the assumption and the program are contradictory.

2. verifying that the machine $((M \angle_{\infty} \Omega_A) || \Omega_P) \downarrow \{\alpha\}$ emits only empty events (Of course, here, α is the alarm signal of Ω_P).

Modular verification: Any sub-process of a compound system sees the remainder of the system as a part of its own environment. The ability to take the environment into account allows modular verification: Having proved a property about a sub-process, one can use this property in the verification of the remainder of the system. More precisely, let M_1, M_2 be two machines, and let P be a safety property we want to prove about $M_1 || M_2$. Assume another safety property P' has been proven about M_2 alone. Then if $M_1 \angle_{\infty} \Omega_{P'}$ satisfies P, so does $M_1 || M_2$. This amounts to considering M_2 as the environment of M_1. Of course, assumptions about the global environment can also be taken into account. With a little additional hypothesis (see [AL89] and the "decomposition theorem" of [KL93]), which amounts to the absence of causality problems, one can even use a seemingly circular reasoning, which consists first in proving a property P_2 of M_2 under the assumption that M_1 satisfies P_1, and then in proving that M_1 satisfies P_1 assuming M_2 satisfies P_2.

Inductive proofs: Moreover, the modular verification technique can be extended to the inductive verification of regular networks of processes [WL89, HLR92a]. Assume one wants to prove a safety property P of the machine $\underbrace{M||M||\ldots||M}_{n \text{ times}}$ for any $n \geq 1$. This can be done by finding a property P' such that

1. M satisfies P'
2. $(M \angle_{\infty} \Omega_{P'})$ satisfies P'
3. P' implies P

(1) proves that P' holds for $n = 1$, (2) proves that, if P' holds for n, then it holds for $n + 1$. So, P' holds for any n, and from (3), so does P. Point (3) can be established by proving that the machine $\chi(I, O) \angle_{\infty} \Omega_{P'}$ satisfies P, where

$$\chi(I, O) = (\{q\}, q, I, O, \{q\} \times E_I \times E_O \times \{q\})$$

is the "chaos" machine which completely non deterministically returns any event of E_O whatever be its input event from E_I. Of course, as for modular verification, a crucial problem is the choice of the property P'. It is considered in the next section.

4 Specification synthesis

Let us come back to modular verification: Given two machines M_1 and M_2, and a safety property P on $S = I_{M_1} \cup O_{M_1} \cup I_{M_2} \cup O_{M_2}$, one must find a property P' on $S_2 = I_{M_2} \cup O_{M_2}$ such that

1. M_2 satisfies P', and
2. $M_1 \angle_{\infty} \Omega_{P'}$ satisfies P

Moreover, the proof of each of the abov? points is expected to be easier than the global proof that $M_1 || M_2$ satisfies P.

This section deals with the synthesis of such a property P', satisfying the point (2) above by construction, when all the involved machines are finite state.

First, we need some additional definitions: Let $\tau = (e_1, e_2, \ldots, e_n, \ldots)$ be a trace on S. We define the ***projection*** of τ on a set S' of signals to be the trace $\tau \downarrow S' = (e_1 \cap S', e_2 \cap S', \ldots, e_n \cap S', \ldots)$. The projection on S' of a set T of traces is $T \downarrow S' = \{\tau \downarrow S' \mid \tau \in T\}$. If T is a set of ***finite*** traces on S, we note $\mathcal{C}(T)$ the set of traces on S which do not have any prefix in T. Obviously, $\mathcal{C}(T)$ is a safety property.

The intuitive method to find P' is the following: Replace M_2 by the "chaos" machine $\chi(I_{M_2}, O_{M_2})$. If $M_1 || \chi(I_{M_2}, O_{M_2})$ satisfies P, the machine M_2 does not influence the satisfaction of P (i.e. we can take $P' = \mathit{true}$) and we are done. Otherwise, $M_1 || \chi(I_{M_2}, O_{M_2})$ can reach some erroneous states, and the role of M_2 is to forbid the traces leading to those states. But, for doing so, M_2 can only restrict its own signals (P' cannot involve signals that M_2 cannot see).

More precisely: Compute $\mathit{Rea}(M_1 || \Omega_P)$. If it does not intersect $Q_{M_1} \times \{q_\alpha\}$, let $P' = \mathit{true}$. Otherwise let $T_{\mathrm{err}} = \mathit{traces}(Q_{M_1} \times \{q_\alpha\})$ be the set of erroneous traces. The following proposition states that $\mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$ is a necessary and sufficient property that M_2 must satisfy so that $M_1 || M_2$ satisfies P:

Proposition: Let $P' = \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$. Then $M_2 \models P' \iff M_1 || M_2 \models P$.

Proof: Let $\tau[n]$ denote the nth prefix of a trace τ.
($\Longrightarrow$): If $M_2 \models P'$, then every trace τ of $M1 || M2$ satisfies $\tau \downarrow S_2 \in \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$. So, $\forall n, (\tau \downarrow S_2)[n] \notin T_{\mathrm{err}} \downarrow S_2$, and since $(\tau \downarrow S_2)[n] = (\tau[n] \downarrow S2), \forall n, \tau[n] \notin T_{\mathrm{err}}$. This means that $\tau \in P$.
($\Longleftarrow$): Assume M_2 does not satisfy P', and let τ be a trace of M_2 not belonging to P'. Then, there exists n such that $\tau[n] \in (T_{\mathrm{err}} \downarrow S_2)$, and there exists a trace $\tau' \in T_{\mathrm{err}}$ such that $\tau[n] = (\tau'[n]) \downarrow (S_2)$. So, the finite trace $\tau'[n]$ is compatible with both M_1 and M_2 and leads to the violation of P. □

Remark: $P' = \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$ is stronger than $P'' = \mathcal{C}(T_{\mathrm{err}}) \downarrow S_2$. A trace τ of M_2 can be the common projection of two traces τ' and τ'' of $M_1 || M_2$, with $\tau' \in \mathcal{C}(T_{\mathrm{err}})$ and $\tau'' \notin \mathcal{C}(T_{\mathrm{err}})$. In that case, τ belongs to P'' (as the projection of τ') and not to P'.

Stronger specifications: However, the necessary and sufficient property $P' = \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$ is sometimes too complicated to be interesting: As a matter of fact, an observer of P' can be as complicated as $M_1 || \Omega_P$. In that case the proof that M_2 satisfies P' is not easier than the proof that $M_1 || M_2$ satisfies P, so nothing is gained with modular proof. Now, any stronger property than P' can be tried. Such a stronger property P'' will still ensure that $M_1 \angle_{\infty} \Omega_{P''}$ satisfies P, but, since it is no longer a necessary property, one cannot conclude that $M_1 || M_2$ does not satisfy P if M_2 does not satisfy P'' .

The basic technique to build such a stronger property P'' is the following: Let us note *avoid* the function $\lambda T.\mathcal{C}(T \downarrow S_2)$. Thus, $P' = \mathit{avoid}(T_{\mathrm{err}})$. Then, for any set T of traces containing T_{err} (i.e., for any upper approximation of T_{err}), $\mathit{avoid}(T)$ is stronger than P'.

5 Module synthesis

In the preceding section, we have outlined a method to find a property P' such that, for any machine M_2 satisfying P', $M_1 || M_2$ satisfies P. P' has been only deduced from M_1 and P, so, it could be built even before M_2 is designed. So, the next question is: can M_2 be *synthesized* from P', considered as a specification? In the finite state case, this is theoretically possible: The specification must be strengthened to become *executable*. P' has been constructed so as to concern only the input/output signals of M_2. Now, an additional constraint is that M_2 must preserve P' by controlling only its output signals. In each reachable state, and whatever be the received input event (possibly satisfying an input assumption), M_2 must be able to perform a transition preserving P'.

Executability: A property P on a set of signals $S = I \cup O$ is *executable* with respect to (I, O), if and only if for any finite trace $\tau \in P$, for any input event $i \in E_I$, there exists an output event $o \in E_O$ such that $\tau.(i \cup o) \in P$. For any safety property P, there exists a weakest executable safety property, implying P. It will be noted $\mathcal{E}(P)$.

Relative precondition: Let P be a safety property on $I \cup O$ and Ω_P be an observer of P. For any $X \subseteq Q_{\Omega_P}$, we define

$$pre^I_{\Omega_P}(X) = \{q \mid \forall i \subseteq I, \exists o \subseteq O, \delta^Q_{\Omega_P}(q, i \cup o) \in X\}$$

In other words, $pre^I_{\Omega_P}(X)$ is the set of states which can lead into X (in one step) *whatever be the input event* received in these states.

Executable strengthening: Let $Exe = \nu X.pre^I_{\Omega_P}(X) \setminus \{q_\alpha\}$. Then Exe does not contain the erroneous state q_α, and

$$\forall q \in Exe,\ \forall i \subseteq I,\ \exists o \subseteq O, \text{ such that } \delta^Q_{\Omega_P}(q, i \cup o) \in Exe$$

Moreover, Exe is the largest set of states satisfying this property. As a consequence, a restriction of Ω_P which detects any trace going out of Exe is an observer of $\mathcal{E}(P)$. Another consequence is that $\chi(O, I)/\Omega_{\mathcal{E}(P)}$ is the most general reactive machine satisfying P. Notice that Exe can be empty, which means that P is not *realizable* in the sense of [ALW89]: There is no machine on (I, O) preserving P against any environment.

Conclusion

Many ideas that have been presented are specializations and simplifications of previous works. For instance:

- The specification of properties by means of a synchronous observer is very close to the approach of COSPAN [Kur89], which takes also into account liveness, both in the program and the properties.
- Several verification approaches take into account the environment, e.g., [Jos87] [AL89] [Jos92], and some of them propose modular methods. The "don't care sets" considered in hardware design and verification [BBH+88, DD92] are also a way of expressing assumptions about the environment.
- The synthesis problems considered in Sections 4 and 5 have been dealt with in several papers — both in Control theory [RW87, RW89, HWT92], and in computer science [PR89, ALW89] — and often extended to cope with liveness properties.

Our simplifications consist in considering *safety properties* of *synchronous systems.* They are suggested by the application field we have in mind: The synchronous model has been shown to be very convenient for the design of reactive systems. In general, most liveness properties are introduced for one of the following reasons:

- To abstract a real-time constraint: For instance, one replace a deadline property by the requirement that something "eventually occurs". Now, in reactive systems, such real-time constraints may not be abstracted, in general: the constraint "*an alarm must be sent within 2 milliseconds after the detection of a dangerous situation*" may not be replaced by "*the alarm must eventually occur*"!
- To restrict the asynchronous semantics: In asynchronous models, concurrency is modelled by non-deterministic interleaving, and this non-determinism must be restricted by fairness constraints. Obviously, this problem does not exist in the synchronous model. In asynchronous systems, compositionality is also achieved by allowing arbitrary (but fair) "stuttering" of processes. The synchronous model is obviously compositional thanks to zero-time, simultaneous, reactions.

Now, these simplifications are certainly fruitful, from a practical point of view. They increase the performances of the automatic tools: For instance, for finite state methods, the synchronous model drastically reduces the size of the considered state graphs; safety properties can be checked by a graph traversal, without storing any path. To specify a safety property by means of an observer, one doesn't need to use — and to learn - any other language than the programming language used to write the program. All these ideas are under implementation in the LUSTRE-SAGA software development system [HCRP91], and actual industrial experimentations are going on.

References

[AB84] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *TCS*, 30, April 1984.

[AL89] M. Abdi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness.* LNCS 430, Springer Verlag, May 1989.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *16th ICALP*, pages 1-17. LNCS 372, Springer Verlag, July 1989.

[BBH+88] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multilevel logic minimization using implicit don't cares. *IEEE Transactions on CAD/ICAS*, CAD-7(6):723–739, June 1988.

[BFH+92] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[BRdSV90] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble.* LNCS 407, Springer Verlag, 1990.

[BS91] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, January 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. Research Report LIX/RR/92/08, Ecole Polytechnique, March 1992. (to appear in the Journal of Logic Programming, special issue on Abstract Interpretation).

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[DD92] M. Damiani and G. DeMicheli. Don't care set specifications in combinational and synchronous logic circuits. Technical Report CSL-TR-92-531, Computer Systems Laboratory, Stanford University, 1992.

[Hal93a] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Int. Workshop on Computer Aided Verification, Elounda (Crete)*, July 1993.

[Hal93b] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLR92a] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7), 1992.

[HLR92b] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[Hol87] G. J. Holzmann. Automated protocol validation in ARGOS : Assertion proving and scatter searching. *IEEE Trans. on Software Ingineering*, SE-13(6):683–696, June 1987.

[HWT92] G. Hoffmann and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *American Control Conference, Chicago*, June 1992.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Jos87] B. Josko. MCTL - An extension of CTL for modular verification of concurrent systems. In *Workshop on Temporal Logic in Specification, Manchester*. LNCS 398, Springer Verlag, 1987.

[Jos92] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29, February 1992.

[KL93] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Fifth Int. Workshop on Computer Aided Verification, Elounda (Crete)*, July 1993.

[Kur89] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*. LNCS 430, Springer Verlag, May 1989.

[LY92] D. Lee and M. Yanakakis. Online minimization of transition systems. In *i24th ACM Symp. on the Theory of Computing, STOC'92, Vancouver, B.C.*, 1992.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92, Stony Brook*. LNCS 630, Springer Verlag, August 1992.

[Mil81] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983.

[Pnu92] A. Pnueli. How vital is liveness? Verifying timing properties of reactive and hybrid systems. In *CONCUR'92, Stony Brook*. LNCS 630, Springer Verlag, August 1992.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *16th Conference on Principles of Programming Languages*. ACM, 1989.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1), January 1987.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), January 1989.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

Contraints in Term Algebras
(Short Survey)

Hubert Comon[*]

May 16, 1993

Unification, which consists in solving equations in the (free) term algebra, is known to be a fundamental operation in many areas of computer science and, in particular, in logic programming. *Disunification*, which consists in solving more complex formulae in the (free) term algebra, also revealed to be a fundamental operation (see [24, 11] for surveys on unification and disunification respectively). Recently, these computations have been seen as *constraint solving* in term algebras and this point of view is actually fruitful. Let us first make clear what we mean by "constraint".

1 Constraints: a definition

A *constraint system* is defined by a *logical language* C (which is in general a fragment of a first-order language), a *structure* M in which the formulae of C are interpreted and an algorithm which decides, for every $\phi \in C$, whether ϕ is satisfiable in M or not. There are many examples: C can be a full first-order language, in which case, the third condition implies the decidability of the (first-order) theory of M. For example, the constraint system could correspond to Presburger arithmetic or the theory of real numbers. It could also be the theory of finite trees, since this theory has been shown decidable [30, 29, 15]. Many other examples will be given later.

Now constraints can be (and have been) studied for their own mathematical interest. But, they can also be used to *constrain* other formulae. More precisely, given a logical language $\mathcal{L}$, a class of structures $\mathcal{M}$ and a satisfaction relation $\models$ on one hand, and a constraint system (C, M, Φ) on the other hand, given in addition, for each structure S in $\mathcal{M}$, an application H_S from the domain D of M into the domain D_S of the structure S, the *constrained logic* consists in

- the language of pairs of formulae (called *constrained formulae*) $\phi|C$ where $\phi \in \mathcal{L}$ and $C \in C$
- a satisfaction relation defined as follows. Given an assignment σ of the free variables of C into D and an assignment θ of the free variables of ϕ into D_S,

$$\sigma, \theta, S \models \phi|C \quad \text{iff} \quad \begin{cases} M \models C\sigma \\ H_S(\sigma) \circ \theta, S \models \phi \end{cases}$$

where $H_S(\sigma)$ is the assignment which associates each variable x in the domain of σ with $H_S(x\sigma)$.

This definition is a bit complicated (and is not satisfactory in many respects), but everything collapses when we consider constraints in *term structures* (also called *symbolic constraints*). Indeed, assuming that M is a term structure and that terms of C are also terms of $\mathcal{L}$, H_S can be (and will

[*] CNRS and LRI, Bat. 490, Université de Paris Sud, 91405 ORSAY cedex, France. E-mail comon@lri.lri.fr

be) chosen as the interpretation defined by S. This means that we do no longer need $\mathcal{M}$ to define the meaning of a constrained formula:

$$\phi|C \text{ represents } \{\phi\sigma \mid \sigma, M \models C\}$$

a constrained formula is a shorthand for the (infinite) set of its instances corresponding to assignments of its free variables which satisfy the constraint. For example, $P(x,y)|x \neq y$ could represent the set of all formulae $P(t_1,t_2)$ where t_1 and t_2 are two distinct terms. This justifies the use of the symbol | which can be read "such that": its use here does not differ from its use in set definitions (in the comprehensive axiom).

Let us conclude these definitions with two remarks: first, this notion of "constraints" is coherent with what is used in practice in logic programming or artificial intelligence, but is quite different from the "constraints" which are used in the algebraic specification community. Secondly, let us emphasize that the constraints are different from what is usually called a "condition"; consider for example the system

$$\begin{cases} f(x) = a \mid a = b \\ a = b \end{cases}$$

where a, b are two distinct constants. Considering these formulae as constrained ones, in which the equality symbol is freely interpreted, the first constrained formula represents the set $\{f(x)\sigma = a\sigma \mid \sigma \models a = b\}$. But $a = b$ is unsatisfiable in the free algebra since a and b are distinct. Hence, this set is empty and the system collapses to the single equation $a = b$. On the contrary, if | is seen as an implication $\leftarrow$, then it is possible to prove $a = b$ using the second equation, and hence, using a cut, we prove that $f(x) = a$.

2 On the use of constraints

It should be quite clear from the definition that (symbolic) constraints can enhance the expressiveness of a logical language, since they allow for a schematization of a possibly infinite sets of formulae. This ability has been used in many situations:

- in constraint logic programming (e.g. [23])
- in order to construct (counter)models [6]
- to forbid particular instances [25]
- to express control strategies in the formulae themselves [34]
- to avoid the combinatorial explosion of semantic unification (e.g. [18])

Constraints might also be used in order to clearly separate irrelevant (from the computational point of view) parts of a formula. This is the case for equational constraints and the so-called *basic strategy* (see [2, 33] for recent developments).

Finally, everybody knows that even if, according to Church's thesis, every programming language has the same expressive power, there are some languages that are better suited than others to the implementation of some algorithms. Similarly, depending on the problem, some logical languages are better suited than others. Constraints provide with the desired flexibility since they are use *combined* with a logical language; it is therefore possible to use any adequate language to express properties of a particular domain.

3 Examples of symbolic constraints

3.1 Equations

The most well-known example of symbolic constraints is unification problems. In such a constraint system, the logical language consists of (disjunction of existentially quantified) conjunctions of equations between terms. The equations are interpreted in the free term algebra $T(F)$ (this is the classical interpretation) or in some quotients $T(F)/_{=_E}$ by a finitely generated congruence $=_E$. Using these constraints in logic programs or automated deductions prevents applying substitutions which may be an expensive operation in case of duplications. That is why they are used since the very beginning in logic programming. In case of interpretations in quotient algebras, equations are also more relevant than unifiers since there might be a very large minimal complete set of unifiers (doubly exponential w.r.t. the size of the equations, in the case of associative-commutative function symbols), whereas the satisfiability of an equation system is much simpler (NP-complete in the case of AC symbols) [27]. We cannot survey all equational theories $=_E$ for which unification is decidable. See [24] instead.

3.2 Equational formulae

More generally, *equational formulae* are arbitrary first-order formulae over an alphabet F of function symbols and the equality predicate symbol. Assuming that they are interpreted in the free term algebra, there are several decision techniques which lead to complete axiomatizations of the algebra of finite trees (see [30, 28, 29, 15, 31] and others). This axiomatization differs, depending on the finiteness of F: when F is finite, the complete axiomatization consists of what is known as "Clark's axioms of equality" plus the *domain closure axiom*

$$\forall x, \bigvee_{f \in F} \exists \vec{x_f}. x = f(\vec{x_f}).$$

Equational formulae can be generalized in various directions. One of them consists in adding *sort constraints*, i.e. an (infinite) family of membership predicate symbols $\in \zeta$ which are interpreted as recognizable subsets of the term algebra $T(F)$. The satisfiability of equational formulae remains decidable with this additional construction [8]. These formulae have been used for solving problems in rewriting theory (e.g. "sufficient completeness" and "inductive reducibility" [11, 8]), and as a constraint system in automated theorem proving [6]. Other applications are described in [11].

The first-order theory of a quotient algebra $T(F)/_{=_E}$ quickly becomes undecidable: a single associative symbol suffice [35], or an associative-commutative symbol [37]. Decidability results include the case where E is a set of flat permutative axioms [30], ground axioms [10] and E is a set of *shallow* equations, a class which encompasses the two previous ones [14].

3.3 Ordering constraints

We already mentioned *ordering contraints* as a mean for expressing ordered strategies. Here, the logical language consists of purely existential formulae, using a set of function symbols F and the two predicate symbols $=$ and $\geq$. Several interpretations of the ordering have been considered:

- Venkataraman in [38] interprets $\geq$ as a subterm ordering, showing the decidability of the system (and undecidability of the first-order theory). However, such an ordering is useless for applications in rewriting theory, since it is not compatible with the term algebra structure.
- The adequate orderings for the applications in automated deduction are the *reduction orderings* (see [16]) which are total on ground terms. A typical example of such an ordering is the *lexicographic path ordering* extending a total precedence, whose existential fragment has been shown

decidable [9]. This result has been extended to other total recursive path (quasi-)orderings [26]. The decidability of the full first-order theory of these orderings is an open question (problem 24 in [17]).

- The theory of partial recursive path orderings appears to be even more difficult; the Σ_4 fragment has been shown undecidable [37]. The decidability of the existential fragment of any such ordering is open. The only hint for this problem is the recent result of [5]: the *positive* existential fragment of the theory of tree embedding is decidable. (Tree embedding is the most simple recursive path ordering: it is the intersection of all simplification orderings).
- Interpreting $\geq$ as *encompassment* (a term t encompasses u if there is an instance of u which is a subterm of t), it is possible to express some properties such as *inductive reducibility* or (sometimes) *sufficient completeness* using first order formulae (see [7]). The first-order theory of a finite number of unary predicate symbols of the form $\geq t_i$ has been shown decidable in [7].

3.4 Set constraints

Many other symbolic constraints have been studied. But it is a too long work to list all of them. Let us conclude with *set constraints* for which many recent beautiful results have been obtained. (And there is still some work to do!). A *set expression* is built from a finite alphabet of function symbols, set variables and the intersection, union an complement symbols. Then, set constraints are finite conjunctions of formulae $e \subseteq e'$ where e, e' are set expressions. These formulae are interpreted assigning set variables to subsets of the term algebra $T(F)$.

Such constraints have been used for the analysis of logic and functional programs (see [21, 1, 19]). The case of *definite* constraints has been solved in [21] and the general case has been further studied by quite different means in [1, 3, 20]. There are two extensions which are still under investigation: adding negative constraints of the form $e \not\subseteq e'$ and adding the *projection* construction, which consists roughly of the inverse of applying a function symbol (see [21]). These extensions have been conjectured decidable.

4 Constraint solving

To *solve* a constraint not only means to decide its satisfiability. More precisely, a constraint solving algorithm is specified by:

- the constraint system (C, M, Φ)
- a subset S of C called the *set of solved forms*

S has to fulfill some requirements (see [11]), in particular, every formula of C should be equivalent (in M) to a solved form, and every solved form should be trivially satisfiable or trivially unsatisfiable. However, there is still some room for choosing the solved forms. For example, in the case of unification (in free algebras), one can choose *tree solved forms* or *DAG solved forms* as explained in [24].

Once solved forms have been specified, we systematically designed constraint solving algorithms using rewriting techniques; we give a set of rewrite rules on formulae, prove their correctness (every formula in C is rewritten to an equivalent formula w.r.t. M), termination (*any* rewriting sequence is finite) and completeness (every irreducible formula is a solved form). There are several advantages for this method:

- The rules can be redundant (and this is actually a desirable property). Then the termination proof might be complex, but it "factorizes" the termination proof for all algorithms obtained by determinizing the control. For example (as we will see below) tree solved forms (Robinson's unification algorithm [36, 22]) and DAG solved forms (corresponding to Martelli and Montanari's unification algorithm [32]) are obtained by strengthening the control on the same set of rules.

- There is a feed-back on the theory, since the rewrite rules are actually an axiomatization of M (see [11])
- the constraint solving algorithms are automatically *incremental* in the following sense: in order to solve $\phi \wedge \psi$, it is possible to use the result of solving ψ.
- We expect to use rewriting tools for proving termination of the constraint solving rules, as we try to show in the following example.

A toy example

We consider the classical unification problems: formulae are conjunctions of equations between terms; they are interpreted in the free term algebra $T(F, X)$. The equality symbol is considered as symmetric (i.e. there is no difference between $s = t$ and $t = s$).

Given a conjunction of equations ϕ, the *occur-check relation* $\geq_\phi$ is the relation on the free variables of ϕ defined as the smallest reflexive-transitive relation which contains $x \geq_\phi y$ as soon as there is an equation $x = t[y]$ in ϕ. (See [16] for the notations on terms and equations that are used here). A variable is *solved* in ϕ if it has only one occurrence, as a member of an equation of ϕ. Let $U(\phi)$ be the set of unsolved variables of ϕ.

Now consider the scheme of rules for unification given below:

Decompose	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	$\rightarrow$	$s_1 = t_1 \wedge \ldots \wedge s_n = t_n$	
Coalesce	$x = y \wedge \phi$	$\rightarrow$	$x = y \wedge \phi\{x \mapsto y\}$	If $x \neq y$ and $x, y \in U(\phi)$
Clash	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$	$\rightarrow$	$\perp$	If $f \neq g$
Eliminate	$x = s \wedge P$	$\rightarrow$	$x = s \wedge P\{x \mapsto s\}$	If $x \in Var(P)$, $x \notin Var(s)$ and $s \notin X$
Check*	$x_1 = t[x_2]_{p_1} \wedge \ldots \wedge x_n = t[x_1]_{p_n}$	$\rightarrow$	$\perp$	If there is an i such that $p_i \neq \Lambda$
Trivial	$s = s$	$\rightarrow$	$\top$	
Merge	$x = s \wedge x = t$	$\rightarrow$	$x = s \wedge s = t$	

If Decompose, Check* do not apply and x is maximal w.r.t. $\geq_\phi$ among the variables occuring at least twice as a member of an equation

Note that in these rules, we relax the classical condition on the sizes in the merge rule (see [24]) and put instead a condition of maximality on x and assume the system decomposed. Whether these conditions can be relaxed without loosing termination was stated as open problem 39 in [17]. We also assume here that there are structural rules for $\wedge$: $\perp \wedge P \rightarrow \perp$, $\top \wedge P \rightarrow P$ and $P \wedge P \rightarrow P$. Moreover $\wedge$ is assumed to be associative and commutative.

The rule system is terminating (modulo the associativity-commutativity of $\wedge$ and the commutativity of $=$). For, consider the *associative path ordering* [4] on formulae, extending the precedence on $F \cup X$ defined by:

- every variable[1] is larger than any function symbol
- every function symbol is larger than $=$ which is in turn larger than $\wedge$
- variables are compared according to the occur check relation

[1] Be careful that variables of the unification problem are seen as constants in the rewriting process! Only the *logical* variables cann be instanciated.

This last statement has to be precised since the occur-check relation actually depends on the formula which is considered. In fact, we consider the (maybe infinite) union of all occur-check relations at any step in the computation. This definition depends on the transformation, but it does not depend on a particular formula ϕ, and we don't need to effectively compute this relation. It may happen that variables are *equivalent* w.r.t. this relation, in which case, they are considered as identical from the associative path ordering point of view.

Note that the associative path ordering has the subterm property and it is monotonic (see [4]). Hence, for proving the termination, we only have to prove that every left hand side of a rule is (strictly) larger than the corresponding right hand side:

- For the structural rules and for **Trivial**, **Check*** and **Clash** the decreasingness is obvious.
- **Decompose** is strictly decreasing because $=> \wedge$ in the precedence and

$$f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) >_{apo} s_i = t_i$$

  by monotonicity and the subterm property.
- **Eliminate** is strictly decreasing because x is strictly larger than the variables of s (it is larger by definition, and it cannot be equivalent to any variable of s since x becomes solved after applying the rule, hence no rule can produce an equation with x in its right hand side). Moreover, variables are larger than function symbols in the precedence, which shows that $x >_{apo} s$.
- **Merge** is strictly decreasing, for the same reason as above: since x is assumed to be maximal in the decomposed system, it cannot be smaller than any variable of s or t, even after further transformations.
- **Coalesce** keeps the problem equivalent w.r.t. $\geq_{apo}$ since x and y are equivalent in the precedence. However, it can only be applied a finite number of times. Hence we can reason modulo this rule, i.e. modulo the strict equivalence on variables.

Now, the system is terminating. If we remove the **Merge** rule, the system is complete w.r.t. tree solved forms and we can find as an instance Robinson's unification algorithm. If we remove the **Eliminate** rule, the system is complete w.r.t. DAG solved forms and we can find an instance of Martelli and Montanari's unification algorithm.

Similar techniques have been applied for the termination proofs of more powerful constraint systems [13, 12].

References

[1] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, CA, 1992.

[2] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction, Saratoga Springs, NY, LNCS 607*. Springer-Verlag, June 1992.

[3] L. Bachmair, harald Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th IEEE Symp. Logic in Computer Science, Montréal*, 1993.

[4] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1(4):329–349, Dec. 1985.

[5] A. Boudet and H. Comon. About the theory of tree embedding. In *Proc. CAAP 93*, 1993. LNCS 668.

[6] R. Caferra and N. Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *Journal of Symbolic Computation*, 13(6):613–642, June 1992.

[7] A.-C. Caron, J.-L. Coquidé, and M. Dauchet. Encompassment properties and automata with constraints. In *Proc. RTA 93*, 1993.

[8] H. Comon. Equational formulas in order-sorted algebras. In *Proc. 17th Int. Coll. on Automata, Languages and Programming, Warwick, LNCS 443*, Warwick, July 1990. Springer-Verlag.

[9] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[10] H. Comon. Complete axiomatizations of some quotient term algebras. In *Proc. 18th Int. Coll. on Automata, Languages and Programming, Madrid, LNCS 510*, July 1991.

[11] H. Comon. Disunification: a survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

[12] H. Comon. Completion of rewrite systems with membership constraints. In W. Kuich, editor, *Proc. 19th Int. Coll. on Automata, Languages and Programming, LNCS 623*, Vienna, 1992. Springer-Verlag. An extended version is available as LRI Research Report number 699, Sept. 1991.

[13] H. Comon and C. Delor. Equational formulas with membership constraints. Technical report, Laboratoire de Recherche en informatique, Mar. 1991. To appear in Information and Computation.

[14] H. Comon, M. Haberstrau, and J.-P. Jouannaud. Decidable properties of shallow equational theories. In *Proc. 7th IEEE Symp. Logic in Computer Science*, Santa Cruz, 1992. Also Research Report 718, Dec. 1991, Laboratoire de Recherche en Informatique, Orsay, France.

[15] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[16] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[17] N. Dershowitz, J.-P. Jouannaud, and J. W. Klop. Open problems in rewriting. Technical report, CWI, Amsterdam, Feb. 1991.

[18] E. Domenjoud. AC unification through order-sorted AC1 unification. *Journal of Symbolic Computation*, 14(6):537–556, Dec. 1992.

[19] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 300–309, 1991.

[20] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *Proc. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS*, 1993.

[21] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[22] J. Herbrand. Recherches sur la théorie de la démonstration. Thèse d'Etat, Univ. Paris, 1930. Also in: Ecrits logiques de Jacques Herbrand, PUF, Paris, 1968.

[23] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. Principles of Programming Languages, Munich*, 1987.

[24] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson.* MIT-Press, 1991.

[25] J.-P. Jouannaud and C. Marché. Termination and completion modulo associativity, commutativity and identity. *Theoretical Comput. Sci.*, 104:29–51, 1992.

[26] J.-P. Jouannaud and M. Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *Proc. 18th Int. Coll. on Automata, Languages and Programming, Madrid, LNCS 510*, 1991.

[27] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on automatic deduction.

[28] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[29] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, pages 348–357, July 1988.

[30] A. I. Mal'cev. Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems. Collected Papers. 1936-1967*, pages 262–289. North-Holland, 1971.

[31] G. Marongiu and S. Tulipani. Decidability results for term algebras. Preprint 9, AILA, 1991.

[32] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[33] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Bruckner, editor, *Proc. European Symp. on Programming, LNCS 582*, pages 371–389, Rennes, 1992. Springer-Verlag.

[34] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction, Saratoga Springs, NY, LNCS 607.* Springer-Verlag, June 1992.

[35] W. V. Quine. Concatenation as a basis for arithmetic. *Journal of Symbolic Logic*, 11(4), 1946.

[36] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[37] R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–458, Nov. 1992.

[38] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *J. ACM*, 34(2):492–510, 1987.

Joining Abstract and Concrete Computations in Constraint Logic Programming *

ROBERTO GIACOBAZZI AND GIORGIO LEVI

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
{giaco,levi}@di.unipi.it

AND

SAUMYA K. DEBRAY

Department of Computer Science, The University of Arizona
Tucson, AZ 85721
debray@cs.arizona.edu

Abstract

In this paper we show how non-standard semantics for constraint-based logic programs (CLP) can be formally specified by means of the same techniques used to define standard semantics. In particular abstract interpretation of constraint logic programs can be viewed as an instance of the CLP framework itself. The use of standard instances of the CLP framework (e.g. $CLP(Bool)$ and $CLP(\mathcal{R})$) for non-standard interpretations, weakens the distinction between concrete and abstract computations in semantics and analysis. We formalize this idea by applying the well known approximation techniques (e.g. the standard theory of closure operators) in conjunction with a generalized notion of constraint system, supporting any program evaluation. The "generalized semantics" resulting from this process, abstracts away from standard semantic objects, by focusing on the general properties of any (possibly non-standard) semantic definition. In constraint logic programming, this corresponds to a suitable definition of the constraint system supporting the semantic definition. Both top-down and a bottom-up semantics are considered.

1 Introduction

Constraint logic programming (CLP) is a generalization of the pure logic programming paradigm, having similar model-theoretic, declarative and operational semantics [23]. CLP is then a general programming paradigm which may be instantiated on various semantic domains. The fundamental linguistic aspect of constraint logic programming is the ability of computing constraints by means

*The work of R. Giacobazzi and G. Levi was supported by the Esprit Basic Research Action 3012 - Compulog and by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. under grants no. 9100880.PF69. The work of S. Debray was supported in part by the National Science Foundation under grants CCR-8901283 and CCR-9123520.

of Horn-like rules. Since this aspect can be separated from the details specific to particular constraint systems, it seems natural to parameterize the semantics of CLP languages with respect to the underlying constraint system. We refer to such a semantics as *generalized semantics* [19]. It turns out that the generalized semantics provides a powerful tool for dealing with a variety of applications relating to the semantics of CLP programs. For example, by considering a domain of "abstract constraints" instead of the "concrete constraints" that are actually manipulated during program execution, we obtain for free a formal treatment of abstract interpretation.

In this paper we show how abstract and concrete interpretations for logic-based languages can be joined into the unifying framework of constraint logic programming. We apply the generalized semantics introduced in [19], intended to generalize the notion of constraint logic programs as firstly introduced in [23]. The algebraic approach we take to constraint interpretation makes it easy to identify a suitable set of operators which can be instantiated in different ways to obtain both standard and non-standard interpretations, relying on some simple axioms to ensure that desirable semantic properties are satisfied. This work has a main technical contributions: to show how a wide class of analysis techniques developed for pure and constraint-based logic programs can themselves be viewed as instances of the constraint logic programming paradigm. This is obtained by considering a general notion of constraint systems which is weak enough to have general applicability and at the same time strong enough to ensure that relevant properties of the standard semantics construction for logic programs are preserved.

The approximation of the meaning of programs by means of relations among the variables involved in the computation is a well known technique to specify a space of approximate assertions for program analysis [15,14]. We argue that the ability of the constraint logic programming paradigm to handle relations on a variety of semantic domains (e.g. real arithmetics, boolean algebras, *etc.*) allows this paradigm to be used in the field of program analysis both as a tool for the formal specification of abstract domains of approximate relations and for the rapid prototyping of static analysis systems. This approach has some interesting practical applications, such as the ability to compile the data-flow analysis directly to an abstract machine for constraint logic programs. This approach, which is a logical extension of the "abstract compilation" scheme discussed in [21], removes the overhead of program interpretation incurred by keeping separate abstract and concrete interpretations, and leads to significant improvements in the speed of analysis [21,32]. Our approach also makes it possible to close the gap that often exists between the formalization of data-flow analyses in terms of abstract interpretation and the realization of efficient implementations by means of appropriate data-structures and efficient algorithms. Applications of our framework to systematically derive efficient algorithms for data-flow analysis (e.g. by means of constraint propagation techniques for constraint solving) have been recently studied in [3].

The paper is structured as follows: in Section 2 we introduce the basic mathematical notations used throughout the paper. Section 3 introduces the main results in [19], thus providing an incremental step-by-step algebraic specification for constraint systems and a top-down and a bottom-up semantics for constraint logic programs which are parametric with respect to the underlying constraint system. In Section 4 we consider generalized semantics for constraint logic programs as a framework for semantics-based analyses for constraint logic programs. An example, namely *ground dependency analysis*, is considered associating *boolean* constraints with standard equations on terms. Some results on approximating constraints by means of *upper closure operators* on constraint systems are also given. This approach points out how some well-known program analysis techniques can be obtained by evaluating an abstract program into a variation of some existing CLP systems, such as $CLP(Bool)$ for ground dependency analysis; and, as shown in

Section 5: $CLP(\mathcal{R})$[1], where a weaker notion of constraint system supporting program analysis is introduced. This is accomplished by focusing on two distinct applications of constraint programming to data-flow analysis, namely: *linear relationships analysis* and *future redundant constraint analysis*. They associate *linear* constraints with standard equations on terms and *range-intervals* with linear constraints on real numbers, respectively. Section 6 contains a survey of the most important related works. Section 7 concludes.

2 Preliminaries

Throughout the paper we will assume familiarity with the basic notions of lattice theory (Birkhoff's text [6] provides the necessary background) and abstract interpretation [12,14]. In the following we recall some basic mathematical notations used in the next sections.

The set of natural numbers, integers, and reals are denoted by $\mathcal{N}$, $\mathcal{Z}$ and $\Re$ respectively. Given sets A and B, $A\backslash B$ denotes the set A where the elements in B have been removed. The powerset of a set S is denoted by $\wp(S)$. The class of finite (possibly empty) subsets of a set S is denoted $\wp^f(S)$. Let Σ be a possibly infinite set of symbols. We denote by Σ^* the family of finite-length strings (sequences) from symbols in Σ, including the empty string Λ. Sequences are typically denoted by $\langle a_1, ..., a_n\rangle$ or simply $a_1, ..., a_n$ for a_i's symbols in Σ. The *length* of s sequence s is denoted $|s|$. The set of objects a_i indexed on a set of symbols Σ is denoted $\{a_i\}_{i\in\Sigma}$. The set of n-tuples of symbols in Σ is denoted Σ^n. When the length of sequences is fixed, sequences and tuples will be often considered equivalent notions. Let $R \subseteq A \times A$ be a binary transitive relation on A, then the transitive closure of R is denoted by R^*. Syntactic identity is denoted $\equiv$.

A *partial ordering* is a binary relation that is reflexive, transitive and antisymmetric. A set P equipped with a partial order $\leq$ is said to be *partially ordered*, and sometimes written $\langle P, \leq\rangle$. A *chain* is a (possibly empty) subset X of a partially ordered set P such that for all $x, x' \in X$: $x \leq x'$ or $x' \leq x$. Given a partially ordered set $\langle P, \leq\rangle$ and $X \subseteq P$, $y \in P$ is an *upper bound* for X iff $x \leq y$ for each $x \in X$. An upper bound y for X is the *least upper bound* iff for every upper bound y': $y \leq y'$; *lower bounds* and *greatest lower bounds* are defined dually. A *complete lattice* is a partially ordered set L such that every subset of L has a least upper bound and a greatest lower bound. A complete lattice L with partial ordering $\leq$, least upper bound $\vee$, greatest lower bound $\wedge$, least element $\bot = \vee\emptyset = \wedge L$, and greatest element $\top = \wedge\emptyset = \vee L$, is denoted $\langle L, \leq, \bot, \top, \vee, \wedge\rangle$. Given partially ordered sets $\langle A, \leq_A\rangle$ and $\langle B, \leq_B\rangle$, a function $f : A \rightarrow B$ is *monotonic* if for all $x, x' \in A$: $x \leq_A x'$ implies $f(x) \leq_B f(x')$. f is *continuous* iff for each non-empty chain $X \subseteq A$: $f(\sqcup_A X) = \sqcup_B f(X)$. A function f is *additive* iff the previous condition are satisfied for each non-empty set $X \subseteq A$ (f is also called *complete join-morphism*). An *upper closure operator* on a partially ordered set $\langle A, \leq\rangle$ is a function $\rho : A \rightarrow A$ that is idempotent, i.e., $\rho(\rho(c)) = \rho(c)$; extensive, i.e., $c \leq \rho(c)$; and monotonic.

To specify function parameters in function definitions we often make use of Church's lambda notation. We write $f : A \rightarrow B$ to mean that f is a total function of A into B. Let $f : A \rightarrow B$ be a mapping, for each $C \subseteq A$ we denote by $f(C)$ the image of C by f: $\{f(x) \mid x \in C\}$. Functions from a set to the same set are usually called *operators*. The identity operator $\lambda x.x$ is often denoted *id*. Let $\langle L, \leq, \bot, \top, \vee, \wedge\rangle$ be a non-empty complete lattice. Let $f : L \rightarrow L$ be a function. The *upper ordinal powers* of f are defined as follows: $f\uparrow 0(X) = X$, $f\uparrow\alpha(X) = f(f\uparrow(\alpha - 1)(X))$ for every successor ordinal α; and $f\uparrow\alpha(X) = \bigvee_{\delta<\alpha} f\uparrow\delta(X)$ for every limit ordinal α. The first limit ordinal

[1] $CLP(\mathcal{R})$ denotes the $CLP(\Re)$ (constraint logic programs on the domain of real numbers) implementation described in [24].

equipotent with the set of natural numbers is denoted by ω.

An *algebraic structure* [20] is a pair $(\mathcal{C}, \mathcal{Q})$ where $\mathcal{C}$ is a non-empty set, called the *universe* of the structure and $\mathcal{Q}$ is a function ranging over a (possibly infinite non-denumerable) index set $\mathcal{I}$ such that for each $i \in \mathcal{I}$, $\mathcal{Q}_i$ are finitary operations on and to elements of $\mathcal{C}$. Algebraic structures are also denoted as $(\mathcal{C}, \mathcal{Q}_i)_{i \in \mathcal{I}}$. In addition to $\mathcal{Q}_i$ operations, some special symbols (e.g. $\otimes$, $\oplus$, $0, \ldots$) will be used to denote algebraic operations, including constants. With an abuse of notation, we will often denote *distinguished elements* of $\mathcal{C}$ as constant operations $\mathcal{Q}_i$ on $\mathcal{C}$. Given algebraic structures $(A, \mathcal{Q}_A)$ and $(B, \mathcal{Q}_B)$ with universes A and B and provided with a common set of basic operators $\mathcal{Q}$, a *(homo)morphism* σ *from* $(A, \mathcal{Q}_A)$ to $(B, \mathcal{Q}_B)$, denoted by $\sigma : (A, \mathcal{Q}_A) \xrightarrow{m} (B, \mathcal{Q}_B)$ is a function $\sigma : A \rightarrow B$ such that: $\sigma(f_A) = f_B$ for each constant symbol in $\mathcal{Q}$ and $\sigma(f_A(a_1, \ldots, a_n)) = f_B(\sigma(a_1), \ldots, \sigma(a_n))$ for each n-ary operation f in $\mathcal{Q}$ and $a_1 \ldots a_n \in A$. Let $(A, \mathcal{Q}_A)$ and $(B, \mathcal{Q}_B)$ as above. Given partially ordered sets $\langle A, \leq_A \rangle$ and $\langle B, \leq_B \rangle$, a *semimorphism* is a function $\sigma : A \rightarrow B$ such that $\sigma(f_A) \leq_B f_B$, for each constant symbol f in $\mathcal{Q}$, and $\sigma(f_A(a_1, \ldots, a_n)) \leq_B f_B(\sigma(a_1), \ldots, \sigma(a_n))$, for each n-ary operation symbol f in $\mathcal{Q}$.

3 Generalized Semantics

As defined in [23], the semantics of constraints is given in terms of an algebraic structure which interprets constraint formulas, while the semantics of a constraint logic program is given in terms of the well known fixpoint, model-theoretic and operational characterizations. In this section we recall some of the basic results on the generalized semantics in [19].

3.1 Term Systems

A *term system* is an algebra of terms provided with a binary operator which realizes substitutions [8]. We are interested in term systems where every term depends only on a finite number of variables (also called *finitary term systems*). They represent the first basic definition in the semantic construction.

Definition 3.1 [term systems [8]]
A *term system* τ is an algebraic structure (τ, S, V) where we refer to the elements of τ as τ-*terms* (*terms* for short); V is a countable set[2] of τ-*variables* (*variables*, for short) in τ; S is a countable set of binary operations on τ, indexed by V; and the following conditions are satisfied, for all $x, y \in V$ and $t, t', t'' \in \tau$:

T_1. $s_x(t, x) = t$, *identity*

T_2. $s_x(t, y) = y$, where $x \neq y$, *annihilation*

T_3. $s_x(t, s_x(y, t')) = s_x(y, t')$ where $x \neq y$, *renaming*

T_4. $s_x(t', s_y(t'', t)) = s_y(s_x(t', t''), s_x(t', t))$ where $x \neq y$ and y *ind* t' *independent composition*

where a τ-term t is *independent* on the τ-variable x, denoted by "x *ind* t", if $s_x(t', t) = t$ for any $t' \in \tau$. We say that a variable v *occurs* in a term t if $\neg(x \textit{ ind } t)$. We denote the set of variables occurring in a term t as $var(t)$. ∎

[2] A more general definition that considers sets of arbitrary cardinalities is given in [8]: for our purposes, it suffices to consider countable sets.

Intuitively, $s_x(t,t')$ denotes the operation "substitute t for every occurrence of the variable x in t'." For notational convenience, we often denote $s_x(t,t')$ as $[t/x]t'$. This notation can be extended to substitutions on multiple variables. Notice that, by T_2, for each $x, y \in V$: x *ind* y iff $x \neq y$. The condition that terms depend on a finite number of variables can be formalized by imposing that the *dimension set* [8]:

$$\left\{ x \in V \,\middle|\, [t/x]t' \not\equiv t' \text{ for some } t \in \tau \right\}$$

is finite for every $t' \in \tau$. A *renaming* of a variable x in a term t is $[y/x]t$ for some $y \neq x$. Standard properties of term systems and substitutions, such as the properties of composition, can be found in [8].

Example 3.1 Let Σ be a finite collection of function symbols. The standard term system $\tau_{(\Sigma,V)} = (T(\Sigma,V), \mathsf{Sub}, V)$ is a term system provided that substitutions in Sub perform standard substitutions. In this case v *ind* t iff the variable v does not occur in t. ◇

Notice that the substitution operators in $\mathcal{S}$ do not perform in general idempotent substitutions.

Definition 3.2
Let Π be a finite collection of predicate symbols and τ be a term system. A (τ,Π)-*atom* has the form $p(t_1,...,t_n)$ where $p \in \Pi$ and $t_i \in \tau$, $\forall i = 1,...,n$. ▮

When clear from the context, we sometimes denote by $\bar{o}$ both a tuple and a set of syntactic objects o (terms, atoms, *etc.*). In particular we denote by $\bar{x}$ a tuple (set) of distinct variables. Let $\bar{o} = (o_1,...,o_n)$ and $\bar{o}' = (o'_1,...,o'_n)$ be tuples (sets) of syntactic objects. We write $\bar{o} \neq \bar{o}'$ to denote $o_i \neq o'_j$ for each i, j.

The following example shows a non-standard instance of the term system algebraic structure.

Example 3.2 Let Σ be a finite set of symbols. Let $\tau_\Sigma = (\wp^f(\Sigma), \mathcal{S}, \Sigma)$, where $\mathcal{S}$ is the family of basic operators s_σ, for $\sigma \in \Sigma$, such that for each $\Delta_1, \Delta_2 \in \wp^f(\Sigma)$:

$$s_\sigma(\Delta_1,\Delta_2) = \begin{cases} (\Delta_2 \setminus \{\sigma\}) \cup \Delta_1 & \text{if } \sigma \in \Delta_2 \\ \Delta_2 & \text{otherwise} \end{cases}$$

In this case, for each $\sigma \in \Sigma$ and finite set $\Delta \subseteq \Sigma$: σ *ind* Δ iff $\sigma \notin \Delta$. τ_Σ is a term system. ◇

3.2 Constraint Systems

We give now a formal algebraic specification for the language of constraints on a given term system. It allows to identify those structures which have to be considered in any non-standard semantic definition. The process of building constraints in any fixpoint evaluation of a given CLP program is mainly based on set-union and conjunction. We want to give an algebraic characterization of this process in order to provide a framework for generalized interpretations of constraint logic programs.

Definition 3.3 [closed semirings [1]]
A *Closed Semiring* is an algebraic structure $(C, \otimes, \oplus, \mathbf{1}, \mathbf{0})$ satisfying the following:

1. $(C, \otimes, \mathbf{1})$ and $(C, \oplus, \mathbf{0})$ are monoids.
2. $\oplus$ is commutative and idempotent.
3. $\mathbf{0}$ is an *annihilator* for $\otimes$, i.e., for every $c \in C$, $c \otimes \mathbf{0} = \mathbf{0} \otimes c = \mathbf{0}$.
4. for any countable sequence of elements $a_1, \ldots, a_n, \ldots$ in C: $a_1 \oplus a_2 \oplus \cdots \oplus a_n \oplus \cdots$ exists and is unique. Moreover associativity, commutativity and idempotence of $\oplus$ apply to countably infinite as well as to finite applications of $\oplus$.
5. $\otimes$ is *left-* and *right-distributive* over finite and countably infinite applications of $\oplus$, i.e., if $C = \{a_1, \ldots, a_n, \ldots\}$ is a countable sequence of elements in C and $c \in C$, then $c \otimes (\oplus C) = \oplus(\{c \otimes c' \mid c' \in C\})$ and $(\oplus C) \otimes c = \oplus(\{c' \otimes c \mid c' \in C\})$, where $\oplus C$ denotes $a_1 \oplus a_2 \oplus \cdots \oplus a_n \oplus \cdots$.

∎

Ioannides and Wong show that the class of relational operators form a closed semiring [22], thus providing a formalization of recursion in the database context. In logic programming, closed semirings summarize, in an algebraic framework, all the aspects dealing with composition of terms like unification and set union. The idea is that of finding the (possibly infinite) set of all paths in the semantic construction. From a semantic viewpoint in fact, each path is a sequence of constraints between vertices in the proof tree. Idempotence, associativity and commutativity are the least set of properties necessary to allow $\oplus$ to model, in the general context of standard as well as non-standard semantics, the "merging" together of information via set union. The operator $\otimes$ corresponds to conjunction of constraints and plays the important role of collecting information during computation. Distributivity allows the representation of constraints as possibly infinite joins of finite meets (also called *simple constraints*). Distributivity plays a fundamental role in the equivalence between the bottom-up and the top-down semantic constructions. Closure on countable sequences of elements in C is necessary to admit constraints that are infinite joins of constraints (this is important in the fixpoint semantic development). A weaker structure, namely a non-distributive closed semiring, will be considered in Section 5.

A semantic definition necessarily implies some notion of "observable behavior": programs that have the same semantics are considered to not be observably different. Modeling the semantics of constraint logic programs in terms of answer constraints corresponds to considering answer constraints as the appropriate observable property [16]. Thus, the notion of solution for a given answer constraint has to be restricted (projected) to the variables of the corresponding query (output variables). Closed semirings are too weak to capture the notion of variable projection. We handle this notion by means of a family of "hiding" operators on the underlying algebra, as in [31]. Cylindric algebras, formed by enhancing Boolean algebras with a family of unary operations called *cylindrifications* [20], provide a suitable framework for this. The intuition here is that given a constraint c, the cylindrification operation $\exists_S(c)$ yields the constraint obtained by "projecting out" information about the variables in S from c. *Diagonal elements* [20] are considered as a way to provide parameter passing [31]. In constraint logic programming the equality symbol "=" is assumed to provide term unification in any constraint system. However, cylindric algebras, which are oriented to first-order languages without function symbols, are not adequate as an algebraic semantic framework for general constraint logic programs, so we extend diagonal elements to deal with generic terms, following the approach in [8]. Finally, for each variable x and term t, a unary operator ∂_x^t extends the substitution operation to idempotent substitutions on constraints.

Definition 3.4 [constraint systems]
A *constraint system* $\mathcal{A}$ is an algebraic structure $(C, \otimes, \oplus, \mathbf{1}, \mathbf{0}, \exists_\Delta, \partial_x^t, d_{t,t'})_{\{x\},\Delta \subseteq V; t,t' \in \tau}$ where C is a set of $\mathcal{A}$-constraints generated by a given set of *atomic constraints*, and is called the *universe* of $\mathcal{A}$; V is a countable set of variables; τ is a term system; $\mathbf{0}, \mathbf{1}, d_{t,t'}$ are distinct (atomic) elements of C, for each $t, t' \in \tau$; $\{\exists_\Delta\}_{\Delta \subseteq V}$ and $\{\partial_x^t\}_{x \in V; t \in \tau}$ are unary operations on C the latter being defined for x *ind* t; $\otimes, \oplus$ are binary operations on C; such that the following postulates are satisfied for any $c, c' \in C$; $\Delta, \Psi \subseteq V$ and $t, t', t'' \in \tau$:

R_1. the structure $(C, \otimes, \oplus, \mathbf{1}, \mathbf{0})$ is a closed semiring;

C_1. $c \oplus \exists_\Delta c = \exists_\Delta c$;

C_2. $\exists_\Delta(c \otimes \exists_\Delta c') = \exists_\Delta(\exists_\Delta c \otimes c') = \exists_\Delta c \otimes \exists_\Delta c'$;

C_3. $\exists_\Delta \exists_\Psi c = \exists_{(\Delta \cup \Psi)} c$;

C_4. $\exists_\Delta$ distributes over finite and countably infinite joins;

D_1. $d_{t,t} = \mathbf{1}$;

D_2. $\exists_{\{x\}} d_{x,t} = \mathbf{1}$;

D_3. $d_{t,t'} = d_{t',t}$;

S_1. $\partial_x^t(c) = \exists_{\{x\}}(d_{x,t} \otimes c)$;

S_2. $\partial_x^t(d_{t',t''}) = d_{[t/x]t',[t/x]t''}$;

S_3. $\partial_x^t(c \otimes c') = \partial_x^t c \otimes \partial_x^t c'$.

∎

With an abuse of notation, when clear from the context, we denote $\partial_x^t(c)$ as $[t/x]c$. The meaning of cylindrification is given by the axioms from C_1 to C_4, while diagonal elements and substitutions are specified by the axioms from D_1 to S_3. Notice that Axiom S_1 and S_2 relate the notion of substitution in the term system τ with diagonal elements of C (which intuitively correspond to the notion of equality constraints) in the expected way. Recursively, a *simple constraint* is any atomic constraint, or the cylindrification (substitution) of a simple constraint, or a finite conjunction (meet) of simple constraints. The notions of "independence" and "occurrence" of variables extend in the obvious way from terms in τ to constraints in C. Let $c \in C$ and $x \in V$: x *ind* c iff $\partial_x^t c = c$ for any $t \in \tau$ such that x *ind* t. A variable x is *bound* in c iff it is existentially quantified in c. A variable x is *free* in c iff $x \in var(c)$ and x is not bound in c. The set of *free variables* in a constraint c is denoted by $FV(c)$. A renaming of c with respect to x is the constraint $\partial_x^y c$ such that $x \neq y$. It is *renamed apart* if also y *ind* c. Let $\{x_1, ..., x_n\} \subseteq V$, in the following we will denote $\exists_{var(c) \setminus \{x_1, ..., x_n\}} c$, i.e. hiding from all the variables in c except $\{x_1, \ldots, x_n\}$, as

$\exists(c)_{\{x_1,\dots,x_n\}}$. We often omit brackets in cylindrifications on sets of variables. We also denote by $d_{(t_1,\dots,t_n),(t'_1,\dots,t'_n)}$ the element $d_{t_1,t'_1} \otimes \dots \otimes d_{t_n,t'_n}$, where $t_1,\dots,t_n,t'_1,\dots,t'_n \in \tau$, and denote $\mathcal{A}$ an arbitrary constraint system $(C,\otimes,\oplus,1,0,\exists_\Delta,\partial^t_x,d_{t,t'})_{\{x\},\Delta\subseteq V;t,t'\in\tau}$. $c_1 \trianglelefteq c_2$ denotes the relation $c_1 \oplus c_2 = c_2$, for $c_1, c_2 \in C$. C is partially ordered by $\trianglelefteq$, and forms a complete lattice.

A number of important properties are shared by constraint systems. In particular, for each $\Delta \subseteq V$, $\exists_\Delta$ defines an additive *upper closure operator* on C[3], while the substitution operator on constraints defines an additive *retraction* on C[4]. Notice that the substitution is not, in general, extensive. Moreover, if x is bound in c then x *ind* c, and if c is a renaming apart of c' with respect to x, then x *ind* c; and if Δ *ind* c then $\exists_\Delta(c \otimes c') = c \otimes \exists_\Delta c'$. An important property on the relation between cylindrification and renaming (with fresh variables) allows us to extend the standard approach to the semantic construction of logic programs to constraint-based programs: $c \otimes \exists_{\{x\}}c = \exists_{\{y\}}(c \otimes \tilde{c}')$ where y *ind* c, c', $y \neq x$ and $\tilde{c}' = \partial^y_x c'$.

Example 3.3 $[CLP(\mathcal{H})]$ Let $\Sigma = \{a,b,\dots,f,g,\dots\}$ be a finite collection of function symbols. Atomic constraints are (one-sorted) equations on the term system $\tau_{(\Sigma,V)}$ (see Example 3.1). The *Herbrand constraint system* $\mathcal{A}_\mathcal{H}$, is the quotient algebra

$$(\wp(\mathcal{E}_\mathcal{H}),\wedge,\cup,true,false,\exists_X,\partial^t_x,\{t=t'\})_{\{x\},X\subseteq V;t,t'\in\tau_{(\Sigma,V)}}/\sim_{EQ},$$

where:

- $\mathcal{E}_\mathcal{H}$ is the set of any finite conjunctions of equations over $\tau_{(\Sigma,V)}$.
- $\mathcal{H}$ is intended to represent the Herbrand interpretation structure, interpreting diagonal elements as unification [23]. A solution θ for a possibly quantified finite conjunction (set) of equations $\exists_X E = \exists_X\{s_1 = t_1,\dots,s_n = t_n\}$ is a grounding substitution for the free variables in E such that there exists a grounding substitution for the bounded variables X: σ, and $s_1\sigma\theta \equiv t_1\sigma\theta$,..., $s_n\sigma\theta \equiv t_n\sigma\theta$. $\mathcal{H} \models E\theta$ denotes that θ is a solution for E [5]. We extend this definition to deal with possibly infinite joins: θ is a solution for $\bigcup_{i\in I} E_i$ iff there exists $i \in I$ such that θ is a solution for E_i[6].
- $\exists$ is the existential quantification which is assumed to be distributive (as well as conjunction) over arbitrary joins: if $X \subseteq V$, θ is a solution for $\exists_X(\bigcup_{i\in I} E_i)$ iff θ is a solution for $\exists_X E_i$ for some $i \in I$.
- For each $c_1 = \bigcup_{i\in I_1} E_i$ and $c_2 = \bigcup_{i\in I_2} E'_i$ denoting possibly infinite joins of (finite) quantified sets of atomic constraints (equations) E_i and E'_i:

$$c_1 \sim_{EQ} c_2 \text{ iff } \bigcup_{i\in I_1}\left\{ \vartheta \,\middle|\, \mathcal{H} \models E_i\vartheta \right\} = \bigcup_{i\in I_2}\left\{ \vartheta \,\middle|\, \mathcal{H} \models E'_i\vartheta \right\}.$$

[3] An *upper closure operator* ρ on a partiall· ordered set $\langle A, \leq\rangle$ is a monotonic, idempotent and extensive (i.e. $\rho(x) \geq x$) operator.

[4] A *retraction* on a partially ordered set $\langle A, \leq\rangle$ is an operator ϱ that is idempotent and monotonic.

[5] This corresponds to the intuitive notion: "a solution assigns values to the free variables of the constraint in such a way that there exists an assignment to the existentially quantified variables such that the constraint is validated" [10].

[6] Thus, in order to handle possibly infinite disjunctions of (finite) sets of equations, we interpret disjunction as set union.

- *true* denotes any constraint having every grounding substitution as a solution while *false* denotes any constraint having an empty set of solutions.
- ∂_x^t, for x not occurring in t, performs idempotent substitutions on constraints, by extending in the obvious way the term substitution notion to constraints.

◇

Example 3.4 [$CLP(\mathcal{R}_n)$] This example formalizes $CLP(\mathcal{R})$ as an instance of our framework. In the following $\vec{x} = (x_1 ... x_n)$ is a vector (point) in $\Re^n$ and x_i is its i-th element. A ***hyperplane*** is the set of points $\vec{x} \in \Re^n$ satisfying $a_1 x_1 + ... + a_n x_n = b$, with not all a's equal to zero. Any hyperplane defines two ***halfspaces*** in the obvious way. A ***convex polyhedron*** is the (possibly unbounded) set of points constituting the intersection of a finite number of halfspaces. Let c be a polyhedron of dimension n, and HS be a halfspace defined by a hyperplane H. If $f = c \cap HS \subseteq H$ then f is called a ***face*** of P. A ***facet*** is a face of dimension $n - 1$. For any finite n, the constraint system of n-dimension linear constraints (the non-linear case is a straightforward extension), denoted by $\mathcal{R}_n$, is: $(\mathcal{P}, \cap, \cup, \Re^n, \emptyset, \partial_x^t, \hat{\exists}_\Delta, [t_1 = t_2])_{\{x\}, \Delta \subseteq V_n; t, t_1, t_2 \in \tau_{Exp}}$, where $V_n = \{x_1, ..., x_n\}$ is a set of n variables, τ_{Exp} is a term system of linear expressions on V_n (a formalization of τ_{Exp} is in Section 5.1) and $\mathcal{P}$ is the set of all space regions in $\Re^n$ defined as possibly infinite unions of ***convex polyhedra*** (each constraint $c \in \mathcal{P}$ can be represented as a possibly infinite set of finite conjunctions of linear equations and disequations on V_n). The ***variable restriction*** operation $\hat{\exists}$ is performed by ***cylindrification parallel to an axis*** [20]: if c is a constraint in $\Re^n$ and $i \leq n$, we define:

$$\hat{\exists}_{x_i} c = \left\{ \vec{y} \in \Re^n \,\middle|\, y_j = x_j \; for \; \vec{x} \in c \; and \; j \neq i \right\}.$$

$\hat{\exists}_{x_i} c$ is the cylinder generated by moving the point set c parallel to the x_i axis. For any two linear expressions $t, t' \in \tau_{Exp}$ and $R \in \{=, \geq, \leq, >, <\}$ we define:

$$[t \; R \; t'] = \left\{ \vec{e} \in \Re^n \,\middle|\, \begin{array}{l} \vec{e} = (e_1, ..., e_n), \\ \theta_{\vec{e}} = [e_1/x_1, ..., e_n/x_n], \\ \theta_{\vec{e}} t \; R \; \theta_{\vec{e}} t' \end{array} \right\}.$$

The substitution operator is: $\partial_x^t = \lambda c. \hat{\exists}_{\{x\}}([x = t] \cap c)$. $\mathcal{R}_n$ is a constraint system. ◇

3.3 Operational Semantics

Constraint logic programming was defined by Jaffar and Lassez to specify relations on a constraint language by means of constraint-based Horn clauses. We follow the approach in [19] by defining Horn-like clauses on constraint systems. Constraint logic programs are defined in the usual way: let $\mathcal{A}$ be a constraint system on the term system τ and Π be a finite set of predicate symbols. An ***$\mathcal{A}$-goal*** is a formula $c \,\Box\, B_1, ..., B_n$, with $n \geq 0$, where c is a simple $\mathcal{A}$-constraint and $B_1, ..., B_n$ is a sequence of (τ, Π)-atoms. An ***$\mathcal{A}$-clause*** is a formula of the form '$H \; :- \; c \,\Box\, B_1, ..., B_n$' where H (the ***head***) is a (τ, Π)-atom and $c \,\Box\, B_1, ..., B_n$ (the ***body***) is an $\mathcal{A}$-goal. If the body is empty, the clause is a ***unit clause***. A ***(generalized) constraint logic program***, also called $\mathcal{A}$-program, is a finite set of clauses. For notational simplicity, we will sometimes omit the superscript from the various semantic functions where the constraint system under consideration is obvious from the context. The family of $\mathcal{A}$-programs is denoted by $CLP(\mathcal{A})$. Finally, the renamings of variables in constraints and terms extend their meaning in the obvious way to any syntactic object (atoms, goals, clauses, programs *etc.*); as well as the notion of independence.

Let $\mathcal{A}$ be a constraint system and $P \in CLP(\mathcal{A})$. Define $\rightsquigarrow_P$ (an *$\mathcal{A}$-derivation step*) to be the least relation on $\mathcal{A}$-goals such that $G \rightsquigarrow_P G'$ iff

- $G = c_0 \,\square\, p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$,
- there exists a renamed version of a clause in P: $p_1(\bar{x}_1)\ :-\ c_1 \,\square\, \bar{B}_1$, such that $var(G) \cap var(\bar{B}_1 \cup \bar{x}_1) = \emptyset$,
- $G' = c_0 \otimes d_{\bar{x}_1,\bar{t}_1} \otimes \exists(c_1)_{\bar{x}_1 \cup var(\bar{B}_1)} \,\square\, \bar{B}_1, p_2(\bar{t}_2), ..., p_n(\bar{t}_n)$,

An $\mathcal{A}$-derivation from an $\mathcal{A}$-goal G is a finite or infinite sequence of $\mathcal{A}$-goals such that every goal is obtained from the previous one by means of a single $\mathcal{A}$-derivation step. A successful derivation is a finite sequence whose last element has an empty body. The operational semantics is then defined in terms of the *success set*, namely the set of successful computations specified by the transitive closure of the transition relation on atomic $\mathcal{A}$-goals, where ε denotes the empty sequence of goals:

$$\mathcal{O}^{\mathcal{A}}(P) = \left\{\, p(\bar{x})\ :-\ \bigoplus \exists(c)_{\bar{x}} \,\middle|\, 1 \,\square\, p(\bar{x}) \rightsquigarrow_P^{*} c \,\square\, \varepsilon \ \right\}.$$

The top-down semantics, defined by the previous transition system, characterizes the descendant (partial) constraints of the initial goal. This semantirs provides information about call-patterns regardless of whether they succeed, finitely fail or do not terminate.

Goal dependent success set semantics is defined in terms of a function $\mathcal{J}_P$ that yields the computed answer constraint for any $\mathcal{A}$-goal, such that $\mathcal{J}_P(G) = \oplus\{\exists(c)_{var(G)} \mid G \rightsquigarrow_P^{*} c \,\square\ \}$. The following lemma proves the AND-compositionality for the operational semantics of constraint logic programs.

Theorem 3.1 **([19])**
Let $G = c_0 \,\square\, p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$ be an $\mathcal{A}$-goal and $P \in CLP(\mathcal{A})$. $\mathcal{J}_P(G) = c$ iff there exist $p_i(\bar{x}_i)\ :-\ c_i \in \mathcal{O}^{\mathcal{A}}(P)$, such that $\bar{x}_i$ ind G and $\bar{x}_i \neq \bar{x}_j$ for $1 \leq i, j \leq n$, $i \neq j$; and $c = \exists(c_0 \otimes d_{\bar{x}_1,\bar{t}_1} \otimes c_1 ... \otimes d_{\bar{x}_n,\bar{t}_n} \otimes c_n)_{var(G)}$.

Since a constraint system can be non-meet-commutative, it is straightforward to notice that the independence on the selection rule does not hold in general in these semantic characterizations. For this reason we have assumed a left-to-right selection rule.

3.4 Fixpoint Semantics

In this section we define a fixpoint bottom-up semantics which is proved to be equivalent to the operational semantics. We allow constrained atoms into the base of interpretations as suggested in [17]. Each constrained atom '$p(\bar{x})\ :-\ c$' represents the set of instances $p(\bar{x})\vartheta$, where ϑ is a solution of the constraint c. We assume $FV(c) \subseteq var(A)$.

It can be shown that the unfolding of a clause is independent on the variable names used in constrained atoms. This can be expressed in the semantics by a relation $\sim$ that captures the notion of equivalence upto renaming on constrained atoms:

Definition 3.5
Let $\mathcal{B}^{\mathcal{A}}$ be the set of constrained atoms of a constraint system $\mathcal{A}$. Define the binary relation $\sim$ on $\mathcal{B}^{\mathcal{A}}$ as follows: given $A_1 \equiv$ '$p(\bar{x}_1)$:- c_1' and $A_2 \equiv$ '$p(\bar{x}_2)$:- c_2' in $\mathcal{B}^{\mathcal{A}}$, $A_1 \sim A_2$ if and only if there exist "renaming apart" variables $\bar{x}'$, i.e. such that $\bar{x}' \neq \bar{x}_1$ and $\bar{x}' \neq \bar{x}_2$ ($\bar{x}'$ *ind* c_1, c_2); and $\partial^{\bar{x}'}_{\bar{x}_1} c_1 = \partial^{\bar{x}'}_{\bar{x}_2} c_2$. The *$\mathcal{A}$-base of interpretations* is $\mathcal{B}^{\mathcal{A}}/\sim$. ∎

In the remainder of the paper we will be concerned primarily with the quotient structure $\mathcal{B}^{\mathcal{A}}/\sim$, and for notational simplicity, denote this by $\mathcal{B}^{\mathcal{A}}$. Moreover, given a syntactic object o, we denote by $p(\bar{x})$:- $c \lessdot_o I$ a constrained atom $p(\bar{x})$:- c such that $[p(\bar{x})$:- $c]_\sim \in I$ and $\bar{x}$ *ind* o. We extend this to specify tuples of renamed apart syntactic objects.

Just as $\oplus$ expresses the notion of "merging together" the information present in two constraints, the operator $\sqcup$ captures the notion of merging together the information present in two sets of constrained atoms; i.e. the operator $\sqcup : \wp(\mathcal{B}) \times \wp(\mathcal{B}) \longrightarrow \wp(\mathcal{B})$ is defined as follows:

$$I_1 \sqcup I_2 = \{[p(\bar{x}) \text{ :- } \oplus\{\partial^{\bar{x}}_{\bar{x}'} c' \mid p(\bar{x}') \text{ :- } c' \lessdot_{\bar{x}} I_1 \cup I_2\}]_\sim\} \text{ for any } I_1, I_2 \in \wp(\mathcal{B}).$$

The relation $\sqsubseteq$, expressing the notion of a set of constrained atoms containing less information than another, is defined as follows: for any $I_1, I_2 \in \wp(\mathcal{B}), I_1 \sqsubseteq I_2$ iff $I_1 \sqcup I_2 = I_2$.

Definition 3.6
The set of *$\mathcal{A}$-interpretations* $\Im^{\mathcal{A}} \subseteq \wp(\mathcal{B}^{\mathcal{A}})$ is the collection of sets of constrained atoms I such that $I \in \Im^{\mathcal{A}}$ iff $I \sqcup \emptyset = \emptyset \sqcup I = I$. ∎

$\langle \Im^{\mathcal{A}}, \sqsubseteq \rangle$ is a complete lattice. Let $p(\bar{x})$:- $\bigoplus_{j \in W} c_j$ be elements of I, for some fixed (possibly infinite) set of indexes W. For each $j \in W$, c_j represents the set of admissible (i.e., computable in the program) solutions for the predicate symbol p, on the variables $\bar{x}$. As the set of indexes W can be infinite, infinite joins of constraints are allowed in constrained atoms. This "closure" property is modeled by the closure of C on infinite joins, as assumed in any constraint system. Notice that it cannot be specified by first-order formulas. The fixpoint semantics of a program P over a constraint system $\mathcal{A}$, $\mathcal{F}^{\mathcal{A}}(P)$, is defined in terms of a continuous immediate consequence operator in the style of [33], i.e. $\mathcal{F}^{\mathcal{A}}(P) = lfp(\mathcal{T}^{\mathcal{A}}_P) = \mathcal{T}^{\mathcal{A}}_P \uparrow \omega(\emptyset)$, where the mapping $\mathcal{T}^{\mathcal{A}}_P : \Im^{\mathcal{A}} \rightarrow \Im^{\mathcal{A}}$, is defined as follows:

$$\mathcal{T}^{\mathcal{A}}_P(I) = \bigsqcup_{C \in P} \left\{ [p(\bar{x}) \text{ :- } \exists(\bar{c})_{\bar{x}}]_\sim \;\middle|\; \begin{array}{l} C: \;\; p(t) \text{ :- } c \,\Box\, p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n), n \geq 0, \\ \bar{x} \text{ ind } C \text{ and for each } i = 1 \ldots n: \\ \quad p_i(\bar{x}_i) \text{ :- } c_i \lessdot_{C, \bar{x}_1 \ldots \bar{x}_{i-1}} I, \; \bar{x}_i \neq \bar{x} \\ \quad c'_i = d_{\bar{x}_i, \bar{t}_i} \otimes c_i, \\ \bar{c} = d_{\bar{x}, \bar{t}} \otimes c \otimes c'_1 \otimes \ldots \otimes c'_n \end{array} \right\}.$$

The fixpoint semantics construction requires, potentially, only a finite set of variables. This follows from the elementary properties of cylindrification with respect to substitution. Intuitively, the hiding allows to define "local environments" which cannot be influenced by substitution. As a consequence, any hidden variable in each of the c_i can be (re)used outside the scope of the hiding, thus making applicable renamings by means of the same variables. The number of variables needed to compute the semantics depends from the program structure.

Example 3.5 Consider the following program over the Herbrand constraint system:

$$
\begin{array}{lll}
p(x) & :- & x = [].\\
p(x) & :- & x = [h|y] \,\Box\, p(y).
\end{array}
$$

The fixpoint computation for $\mathcal{T}_P$ returns the following interpretation for p (we denote by , and ; conjunction and disjunction (set-union) of constraints):

$$
\begin{array}{lll}
p(x) & :- & x = [];\\
 & & \exists_{h,y,x'}(x = [h|y], y = x', x' = []);\\
 & & \exists_{h,y,x''}(x = [h|y], y = x'', \exists_{h,y,x'}(x'' = [h|y], y = x', x' = []));\\
 & & \exists_{h,y,x'} \left(\begin{array}{l} x = [h|y], y = x', \\ \exists_{h,y,x''} \left(\begin{array}{l} x' = [h|y], y = x'', \\ \exists_{h,y,x'}(x'' = [h|y], y = x', x' = []) \end{array} \right) \end{array} \right);\\
 & & etc....
\end{array}
$$

The set of variables needed to compute the fixpoint is $\{x, x', x'', h, y\}$. ◇

The following result states the equivalence between the operational and the fixpoint semantics, for any constraint system $\mathcal{A}$. We need that V is a denumerable set of variables.

Theorem 3.2 ([19])
Let $\mathcal{A}$ be a constraint system and $P \in CLP(\mathcal{A})$, then $\mathcal{F}(P) = \mathcal{O}(P)/\sim$.

4 Abstract Constraint Systems

The definition of an abstract constraint system, which specifies a non-standard semantics for a constraint programming language, is performed in two steps: *term abstraction* and *constraint abstraction*. In the first step new syntactic objects are introduced to represent concrete terms. In the second one, constraints on the abstracted term system are defined.

In general, a constraint system is an interpretation (in a closed semiring) for constraint formulas. To relate constraint systems, we follow the approach to "static semantics correctness" in [5]. Correctness of non-standard semantic specifications can be handled in an algebraic way through the notion of morphism. However, the algebraic notion of morphism can be made less restrictive by assuming that the carriers of the algebras involved are partially ordered sets. We use a weaker notion of morphism between algebraic structures, capturing the approximation possibly induced by abstract interpretations or by any approximate semantics defined in the framework.

A morphism of term systems, $\kappa : \tau \xrightarrow{m} \tau'$, is a function mapping terms of τ to terms of τ' such that $\forall t_1, t_2 \in \tau$ and $x \in V$: $\kappa(s_x(t_1, t_2)) = s'_{\kappa(x)}(\kappa(t_1), \kappa(t_2))$, where s and s' are the substitution operators in τ and τ' respectively. Let $\mathcal{A}$ and $\mathcal{A}'$ be constraint systems $(\mathcal{A}' = (\mathcal{C}', \otimes', \oplus', \mathbf{1}', \mathbf{0}', \exists'_\Delta, \partial'^{t}_{x}, d'_{t_1,t_2})_{\{x\},\Delta \subseteq V'; t,t_1,t_2 \in \tau'})$ be constraint systems. There exists a semi-morphism $\alpha : \mathcal{A} \xrightarrow{s} \mathcal{A}'$ iff there exists a morphism of term systems $\kappa : \tau \xrightarrow{m} \tau'$ such that for each $c, c_1, c_2 \in \mathcal{C}$, $C \subseteq \mathcal{C}$, $\{x\}, \Delta \subseteq V$ and $t, t_1, t_2 \in \tau$ such that x *ind* t, the following hold:

$$
\begin{array}{l}
\alpha(\mathbf{0}) = \mathbf{0}'\\
\alpha(\mathbf{1}) \trianglelefteq' \mathbf{1}'
\end{array}
$$

$$\alpha(\oplus C) \trianglelefteq' \oplus' \alpha(C)$$
$$\alpha(\exists_\Delta c) \trianglelefteq' \exists'_{\kappa(\Delta)} \alpha(c)$$
$$\alpha(c_1 \otimes c_2) \trianglelefteq' \alpha(c_1) \otimes' \alpha(c_2)$$
$$\alpha(d_{t_1,t_2}) \trianglelefteq' d'_{\kappa(t_1),\kappa(t_2)}$$

Semimorphisms of constraint systems will be often denoted as α_κ. Notice that $\alpha(\partial^t_x c) \trianglelefteq' \partial'^{\kappa(t)}_{\kappa(x)} \alpha(c)$.

Definition 4.1
Let $\mathcal{A}$ and $\mathcal{A}'$ be constraint systems with universes $\mathcal{C}$ and $\mathcal{C}'$ and term systems τ and τ' respectively. $\mathcal{A}'$ is *correct* with respect to $\mathcal{A}$ iff there exists a semimorphism α_κ ($\kappa : \tau \xrightarrow{m} \tau'$ and $\alpha : \mathcal{A} \xrightarrow{s} \mathcal{A}'$) which is a surjective and additive mapping of $\langle \mathcal{C}, \trianglelefteq \rangle$ into $\langle \mathcal{C}', \trianglelefteq' \rangle$. ∎

Additivity and surjectivity allow the semimorphism to associate the "best" approximating constraint in $\mathcal{A}'$ with any concrete constraint in $\mathcal{A}$. As usual, this is captured by the notion of *Galois insertion*, as specified by the following, where a pair of functions (α, γ) is a Galois insertion of $\langle \mathcal{C}', \trianglelefteq' \rangle$ into $\langle \mathcal{C}, \trianglelefteq \rangle$ iff α and γ are monotonic, $\alpha(\gamma(c')) = c'$ and $c \trianglelefteq \gamma(\alpha(c))$ for each $c \in \mathcal{C}$ and $c' \in \mathcal{C}'$ [12,14]. If $\mathcal{A}'$ is correct with respect to $\mathcal{A}$ by means of a semimorphism α_κ, there exists a Galois insertion of $\langle \mathcal{C}', \trianglelefteq' \rangle$ into $\langle \mathcal{C}, \trianglelefteq \rangle$.

In the framework of abstract interpretation, correctness of fixpoint approximations require in addition some conditions on correctness of the non-standard semantics operators [12]. With the assumption of additivity, semimorphisms are adequate to specify both Galois insertions, and correctness of constraint systems. Let $\mathcal{A}'$ be a constraint system which is correct with respect to $\mathcal{A}$, by means of a semimorphism α_κ. Let $P = \{C_1, ..., C_m\}$ be a program in $CLP(\mathcal{A})$. The *corresponding program on* $\mathcal{A}'$ is a set of clauses $\{C'_1, ..., C'_m\}$ such that for each $i = 1, ..., m$ if $C_i = \ p(\tilde{t}) \ :- \ c \ \Box \ p_1(\tilde{t}_1), ..., p_n(\tilde{t}_n)$ then $C'_i = \ p(\kappa(\tilde{t})) \ :- \ \alpha(c) \ \Box \ p_1(\kappa(\tilde{t}_1)), ..., p_n(\kappa(\tilde{t}_n))$ where κ extends in the obvious way on tuples of terms. The following theorem relates the semantics of a program with the (non-standard) semantics of the corresponding program defined on a correct constraint system.

Theorem 4.1
Let $P \in CLP(\mathcal{A})$ and $P' \in CLP(\mathcal{A}')$ be the corresponding program on $\mathcal{A}'$. Assume $\mathcal{A}'$ be correct with respect to $\mathcal{A}$. There exists $\beta : \Im^{\mathcal{A}} \rightarrow \Im^{\mathcal{A}'}$ such that $\beta(\mathcal{F}^{\mathcal{A}}(P)) \sqsubseteq' \mathcal{F}^{\mathcal{A}'}(P')$.

Given a (fixpoint) concrete semantics, data-flow analysis usually requires computing the limit of Kleene chains. Convergence to the least fixpoint can either be obtained by forcing the abstract domain to satisfy the *ascending chain condition* or to use *widening* and *narrowing* operators to accelerate convergence for fixpoint approximations, as suggested in [12]. In the following we consider the conditions on the constraint system that ensure the resulting abstract domain to satisfy the ascending chain condition. We introduce the ascending chain condition on constraint systems and we show how this condition ensures finiteness in fixpoint computations. This approach is more related with the constraint system structure than the widening/narrowing one, which is in turn more related with the fixpoint computation.

A set of constraints $\{c_1, ..., c_n, ..\}$ is said to be *free-variable bounded* iff there exists a finite set of variables $\tilde{V}$ such that $FV(c_i) \subseteq \tilde{V}$ for each $i \geq 1$. The following definition is important for abstract interpretation purposes:

Definition 4.2
A constraint system $\mathcal{A}$ is *Noetherian* iff its universe C does not contain any infinite chain of free-variable bounded constraints. ▮

The free-variable-boundedness condition here is crucial, for otherwise any constraint system with a denumerable set of variables is not Noetherian. To see this, consider the constraints $c_i \equiv X_1 \vee \cdots \vee X_i$: the set of constraints $\{c_i \mid i \geq 1\}$, ordered by entailment, forms an infinite ascending chain even on a two-valued boolean interpretation. However, it is easy to see that this set is not free-variable-bounded. Given a Noetherian constraint system $\mathcal{A}$, the domain $\Im^{\mathcal{A}}$ is Noetherian, and $\mathcal{F}^{\mathcal{A}}(P)$ can be computed by iterating $\mathcal{T}_P$ a finite number of times.

Different semantic characterizations lead to different abstract evaluation strategies. *Top-down* abstract interpretation corresponds to the abstraction of the standard operational semantics. *Bottom-up* abstract interpretation instead allows to compute a finite abstract approximation of the fixpoint semantics associated with a given constraint logic program. Goal-independence is an attractive feature of bottom-up evaluations. Global program analysis, especially useful in type inference, can then be specified as a bottom-up evaluation in a suitable constraint system. In the following we will concentrate on bottom-up (fixpoint-based) abstract interpretations only. This because the possibility of using only a finite set of variables on which renamings are performed is attractive for proving that a constraint system is Noetherian (see for instance Section 5.1).

We illustrate the previous idea by means of a simple example of data-flow analysis for *ground dependences* in pure logic programs [4,11].

Consider the (concrete) term system $\tau_{(\Sigma,V)}$ being defined over a finite set of variables V. Let us consider the term system τ_V as defined in Example 3.2. Terms are finite sets of variables. Ground terms are denoted with the empty set of variables. It is straightforward to notice that *var* is a morphism of term systems.

Marriott and Søndergaard have proposed an elegant domain, named *Prop*, to represent ground dependences among arguments in atoms. This domain can be expressed as an instance of our framework using the algebra of propositional formulas with disjunction. Let $\mathcal{A}_{Prop} = (Prop_V, \wedge, \vee, true, false, \exists_X, \partial^t_x, \wedge(t) \leftrightarrow \wedge(t'))_{\{x\},X\subseteq V;t,t'\in\tau_V\cup\{\emptyset\}}$ be the algebra of possibly existentially quantified disjunctions of formulas, defined on the term system τ_V, by the connectives $\wedge$ and $\leftrightarrow$; where, for each finite set of variables $\{x_1, ..., x_m\} \in \tau_V$: $\wedge(\{x_1, ..., x_m\}) = x_1 \wedge ... \wedge x_m$, and $\wedge(\emptyset) = true$.

Intuitively, the formula $x \wedge y \wedge z \leftrightarrow w \wedge v$ represents an equation $t = t'$ where $var(t) = \{x, y, z\}$ and $var(t') = \{w, v\}$; $x \wedge y$ represents a term whose groundness depends upon variables x and y; while $x \vee y$ represents a set of terms whose groundness depends upon variables x or y. Local variables are hidden by existential quantification, projecting away non-global variables in the computation. Since $x \leftrightarrow true$ is equivalent to x, a variable x instantiated with ground term is denoted x (i.e. the expression x denotes that x is rigid). Substitution is defined in the obvious way. It is easy to prove that, because of the finiteness of V, $\mathcal{A}_{Prop}/\leftrightarrow$ is a finite (and then noetherian) constraint system.

We associate with each equational constraint in $\mathcal{A}_{\mathcal{H}}$, a boolean expression specifying groundness relationships among variables in predicates. The following example shows this technique.

Example 4.1 Consider the following program to reverse a list:

```
nrev([], []).
nrev([H|L], R) :- nrev(L, L1), append(L1, [H], R).

append([], L, L).
append([H|Y], X2, [H|Z]) :- append(Y, X2, Z).
```

The corresponding *Prop* program for groundness analysis is:

$$\begin{aligned}
&\mathit{nrev}(x_1,x_2)\ :-\ x_1 \wedge x_2.\\
&\mathit{nrev}(x_1,x_2)\ :-\ x_1 \leftrightarrow (h \wedge l)\ \Box\ \mathit{nrev}(l, r1), \mathit{append}(r_1, h, r).\\
&\\
&\mathit{append}(x_1,x_2,x_3)\ :-\ x_1 \wedge x_2 \leftrightarrow x_3.\\
&\mathit{append}(x_1,x_2,x_3)\ :-\ x_1 \leftrightarrow (h \wedge y) \wedge x_3 \leftrightarrow (h \wedge z)\ \Box\ \mathit{append}(y, x_2, z).
\end{aligned}$$

The reader may verify that the abstract semantics for append and nrev can be derived by evaluating the modified program in $CLP(\mathcal{A}_{Prop})$ (which corresponds to the standard $CLP(Bool)$). They are given by:

$$\begin{aligned}
&\{\ \mathit{append}(x_1,x_2,x_3)\ :-\ x_3 \leftrightarrow (x_1 \wedge x_2)\}; \text{ and}\\
&\{\ \mathit{nrev}(x_1,x_2)\ :-\ x_1 \leftrightarrow x_2\}.
\end{aligned}$$

which correspond precisely with the behavious of the program with respect to groundness: in append the third argument is ground iff the first two is ground, while in nrev the first argument is ground iff the second is ground. ◇

4.1 The Approximation Operator on Constraint Systems

The space of approximate constraints can be specified using upper closure operators, which formalize the idea of approximation [14]. Idempotence can be interpreted as the fact that all information is lost at once in the abstraction process; extensivity captures the essence of approximation as *weakening*, while monotonicity of the closure states that approximation is order preserving.

In the following we introduce the basic properties of upper closure operators on a constraint system. These properties allow the resulting algebraic structure to be a constraint system as well. The following results extend the classical ones on closure operators [14] to constraint systems. In particular we characterize the approximation induced when α_κ behaves as a morphism of constraint systems. Following this approach, we can extend most of the well known techniques for abstract domain specification to constraint systems.

Definition 4.3
Let $\mathcal{A}$ be a constraint system with universe C. A *compatible upper closure operator* ρ on $\mathcal{A}$ is an upper closure operator on $\langle C, \trianglelefteq\rangle$ satisfying the following properties: for each $c, c' \in C$:

1. $\rho(\exists_\Delta c) = \exists_\Delta \rho(\exists_\Delta c)$; (∃-quasi closure)
2. $\rho(c \otimes c') = \rho(\rho(c) \otimes \rho(c'))$. (⊗-quasi morphism)

Since a compatible upper closure operator is a closure operator, it maps each constraint to one that approximates it. In addition, $\otimes$-quasi morphism relates meets of abstract constraints with meets of concrete constraints (recall that an upper closure operator is also a *quasi-complete join-morphism*, namely for each $D \subseteq C$, $\rho(\bigoplus_{c \in D} c) = \rho(\bigoplus_{c \in D} \rho(c)))$. Finally, the $\exists$-quasi closure property ensures that the approximation of a constraint which is hidden on a set of variables, is still hidden on the same set of variables. From this condition we can prove that ρ satisfies the $\exists$ and ∂-quasi morphism condition (i.e. $\rho(\exists_\Delta c) = \rho(\exists_\Delta \rho(c))$ and $\rho(\partial_x^t c) = \rho(\partial_x^t \rho(c))$, for each $c \in C$, $\{x\}, \Delta \subseteq V$ and $t \in \tau$ such that x *ind* t) and that $\rho \circ \exists_\Delta$ is an upper closure operator.

Notice that $\exists_\Delta \circ \rho$ is not idempotent, unless $\exists_\Delta$ and ρ commute. This is in accordance with a classical result of closure theory saying that any composition of two upper closure operators is an upper closure operator iff they commute [30].

Let $\mathcal{A} = (C, \otimes, \oplus, \mathbf{1}, \mathbf{0}, \exists_\Delta, \partial_x^t, d_{t_1,t_2})_{\{x\},\Delta \subseteq V; t, t_1, t_2 \in \tau}$ be a constraint system and ρ be a compatible upper closure operator on $\mathcal{A}$. We define:

$$\rho(\mathcal{A}) = (\rho(C), \tilde{\otimes}, \tilde{\oplus}, \mathbf{1}, \rho(\mathbf{0}), \tilde{\exists}_\Delta, \tilde{\partial}_x^t, \rho(d_{t_1,t_2}))_{\{x\},\Delta \subseteq V; t, t_1, t_2 \in \tau}$$

where $\rho(C) = \{c \in C \mid c = \rho(c)\}$, for each $c, c_1, c_2 \in \rho(C)$, $\{x\}, \Delta \subseteq V$ and $t \in \tau$ such that x *ind* t: $c_1 \tilde{\oplus} c_2 = \rho(c_1 \oplus c_2)$, $c_1 \tilde{\otimes} c_2 = \rho(c_1 \otimes c_2)$, $\tilde{\exists}_\Delta c = \rho(\exists_\Delta c)$ and $\tilde{\partial}_x^t c = \rho(\partial_x^t c)$.

Theorem 4.2
Let ρ be a compatible upper closure operator on the constraint system $\mathcal{A}$. $\rho(\mathcal{A})$ is a constraint system.

Example 4.2 Cylindrifications are monotonic operators, while idempotence and extensivity are specified by axioms C_4 and C_1 respectively. Moreover, cylindrifications commute thus, if Δ and Ψ are sets of variables and c is a constraint: $\exists_\Delta \exists_\Psi \exists_\Delta c = \exists_\Delta \exists_\Psi c$. However, for each set of variables Δ: $\exists_\Delta$, is not a compatible upper closure operator on the constraint system because it does not satisfy the $\otimes$-quasi morphism condition (see Axiom C_2). ◇

By $\otimes/\oplus$-quasi morphism, $\rho(\mathcal{A})$ is correct with respect to $\mathcal{A}$ by means of the morphism ρ_{id}.

As observed in [14], any Galois insertion (α, γ) defines an upper closure operator $\rho = \gamma \circ \alpha$ on the corresponding (concrete) complete lattice.

Corollary 4.3
Let $\mathcal{A}^a$ be a constraint system which is correct with respect to a constraint system $\mathcal{A}$ by means of a surjective and additive morphism α_κ. Let $\gamma = \lambda c^a . \oplus \{c \mid \alpha(c) \trianglelefteq^a c^a\}$ and $\rho = \gamma \circ \alpha$. $\rho(C)$ is isomorphic to C^a.

Let $\mathcal{A}$ be a constraint system and $\mathcal{A}^a$ be correct with respect to $\mathcal{A}$ by means of a surjective and additive morphism α_κ. Let $\gamma = \lambda c^a . \oplus \{c \mid \alpha(c) \trianglelefteq^a c^a\}$ and $\rho = \gamma \circ \alpha$.

Theorem 4.4
Let $\Delta \subseteq V$ and $c, c_1, c_2 \in C$. Then $\exists_\Delta \rho(\exists_\Delta c) = \rho(\exists_\Delta c)$ and $\rho(\rho(c_1) \otimes \rho(c_2)) = \rho(c_1 \otimes c_2)$.

In the following we study some sufficient conditions on $\mathcal{A}$ and α_κ to let the interpretation of $\otimes$, $\exists$ and ∂ operators be not affected by the closure $\rho = \gamma \circ \alpha$ (i.e. the closure becomes a morphism of $\otimes$, $\exists$ and ∂).

Theorem 4.5
Let $c_1, c_2 \in C$: $\rho(c_1) \otimes \rho(c_2) \trianglelefteq \rho(c_1 \otimes c_2)$. If $\mathcal{A}$ is $\otimes$-idempotent and 1 is the annihilator for $\oplus$, then $\rho(c_1 \otimes c_2) = \rho(c_1) \otimes \rho(c_2)$.

The following theorem gives a sufficient condition on $\mathcal{A}^a$ such that the composition of $\exists$ and ρ is a closure (i.e. $\exists$ and ρ commute).

Theorem 4.6
Let $c, c' \in C$ and $\Delta \subseteq V$. $\exists_\Delta \rho(c) \trianglelefteq \rho(\exists_\Delta c)$. If $\alpha(\exists_\Delta c) = \alpha(\exists_\Delta c') \;\Rightarrow\; \alpha(c) = \alpha(c')$, then $\rho(\exists_\Delta c) = \exists_\Delta \rho(c)$

Notice that the previous condition: $\alpha(\exists_\Delta c) = \alpha(\exists_\Delta c') \;\Rightarrow\; \alpha(c) = \alpha(c')$ state the injectivity of cylindrification in the abstract constraint system.

Theorem 4.7
Let $\mathcal{A}$ be a constraint system and ρ be an upper closure operator on C which commutes with $\exists$, it is a $\otimes$-morphism and for each $t, t' \in \tau$: $\rho(d_{t,t'}) = d_{t,t'}$. Then $\partial^t_x \rho(c) = \rho(\partial^t_x c)$.

We finally give a representation result for abstract constraint systems. Recall that given a partially ordered set $\langle P, \leq \rangle$, $S \subseteq P$ is *convex* iff for each $c, c'' \in S$, $c' \in P$ such that $c \leq c' \leq c''$ then $c' \in S$. It turns out that any (compatible) upper closure approximation of a constraint system defines a partition of the universe of constraints into convex sets of constraints:

Proposition 4.8
Let $\mathcal{A}$ be a constraint system and ρ be an upper closure operator on its universe of constraints C For each $c \in C$ the set $c^\rho = \{c' \in C \mid \rho(c) = \rho(c')\}$ is convex.

As a consequence, the closure of a constraint system $\mathcal{A}$ under a given upper closure operator ρ (i.e. $\rho(\mathcal{A})$) is the algebraic structure of "abstract" constraints each representing a convex space of "concrete" solutions. The axioms for compatible closure operators (i.e. axioms 1 and 2) ensure that $\rho(\mathcal{A})$ is a constraint system.

5 Non-distributive Analysis

Our framework is not appropriate to formalize an interesting class of constraint systems which are proved to be useful for program analysis (e.g. see *linear relationships analysis* below).

Let us consider a (possibly non-compatible) upper closure operator. For the family of constraint systems where $\otimes$ is idempotent and commutative, and 1 is annihilator for $\oplus$, any meet of closed constraints is still closed, i.e., $\rho(c_1) \otimes \rho(c_2) = \rho(\rho(c_1) \otimes \rho(c_2))$. Thus, any compatible upper closure operator is a $\otimes$ morphism (see Theorem 4.5). This assumption is too strong for a wide class of closure operators useful in program analysis. For any upper closure operator ρ: $\rho(c_1 \otimes c_2) \trianglelefteq \rho(c_1) \otimes \rho(c_2)$. The converse does not hold in general. In the following we will consider $\otimes$-idempotent and commutative constraint systems where 1 is annihilator for $\oplus$.

Definition 5.1
A *weakly compatible upper closure operator* on a constraint system $\mathcal{A}$, with universe of constraints C, is an upper closure operator on C such that: $\rho(\mathbf{0}) = \mathbf{0}$, $\rho(d_{t,t'}) = d_{t,t'}$ and $\rho \circ \exists_\Delta = \exists_\Delta \circ \rho$, for any term t, t' and set of variables Δ. ▮

Theorem 5.1
Let $\mathcal{A} = (C, \otimes, \oplus, \mathbf{1}, \mathbf{0}, \exists_\Delta, \partial^t_x, d_{t_1,t_2})_{\{x\},\Delta\subseteq V;t,t_1,t_2\in\mathcal{T}}$ be a constraint system and ρ be a weakly compatible upper closure operator on $\mathcal{A}$. $\rho(\mathcal{A}) = (\rho(C), \otimes, \tilde{\oplus}, \mathbf{1}, \mathbf{0}, \exists_\Delta, \partial^t_x, d_{t_1,t_2})_{\{x\},\Delta\subseteq V;t,t_1,t_2\in\mathcal{T}}$ where $\tilde{\oplus} = \lambda C.\rho(\oplus C)$, is a non-distributive constraint system.

Assume a constraint system $\mathcal{A}$ where the axiom of distributivity is replaced by the weaker relation: $c \otimes (c_1 \oplus c_2) \trianglerighteq (c \otimes c_1) \oplus (c \otimes c_2)$. Distributivity has been assumed to prove the equivalence results between fixpoint and operational semantics. The second one in fact is a kind of "all solutions" semantics, where the join is taken at the end of all the possible computations, while in the fixpoint case, the semantic construction applies the join operator at each partial computation step (an equivalent operational semantics can be easily defined: this would correspond to the bottom-up execution strategy of deductive databases rather than the standard operational interpretation of logic programs [28]). In this case, as the constraint system is not distributive any more, we can only have a further approximation level by applying bottom-up instead of top-down, i.e. $\mathcal{O}(P) \sqsubseteq \mathcal{F}(P)$. In the following we study this class of constraint systems by means of an example.

Example 5.1 The problem of *future redundant constraints* in $CLP(\mathcal{R})$ has been studied in the context of compiler optimization [25]. Intuitively, a constraint in a clause is future redundant if, after testing the constraint for satisfiability, adding or not the constraint to the current computed constraint (also named *store*) does not contribute to the answer constraint. This because, in the computation, stronger constraints are added to the store. This information can be used for a variety of optimizations [25]. In this example we sketch a formalization of this analysis as a non-standard CLP computation using a slightly different notion of redundancy. Consider the constraint system $\mathcal{R}_n$ of Example 3.4. Let $P \in CLP(\mathcal{R}_n)$ and ρ be some extensive operator on $\mathcal{R}_n$ (upper closure operators are appropriate to this purpose). Assume p be a predicate symbol defined in P and let $C = p(\bar{t})\ :-\ \hat{c} \cap c' \,\square\, B \in P$ be a clause defining p. Let $P' = (P \setminus \{C\}) \cup \{p(\bar{t})\ :-\ c' \,\square\, B\}$. If $p(\bar{x})\ :-\ c_p$ is in $\mathcal{F}(P')$, i.e., c_p is the answer constraint for p in the modified program, $c_p \cap \hat{c} \neq \emptyset$ (i.e. $c_p \wedge \hat{c}$ is solvable) and for each convex polyhedron $c \in c_p$: $\rho(c) \subseteq \hat{c}$ (i.e. $\hat{c}$ is weaker than $\rho(c)$), then $\hat{c}$ is future redundant in C. To prove this claim we just note that by ρ-extensivity, for each constraint c: $c \subseteq \rho(c)$.

A suitable choice of an extensive operator on $\mathcal{R}_n$ is provided by approximating any convex polyhedron with a *hypercube*, which is a polyhedron whose facets are parallel to the axes (similar techniques have been used for static array bound checking [12]). For any set of polyhedra $c \in \mathcal{P}$, define $box(c)$ as the least hypercube containing c. box is clearly an upper closure operator on the domain of convex polyhedras ordered by set inclusion. To provide parameter passing, we allow diagonal elements in the abstract domain of constraints. The approximation of diagonal elements by their least hypercubes should correspond in fact to associate the whole space region $\Re^n$ with each equation, thus making the parameter passing useless. Thus, if c is a hyperplane, $box(c) = c$. Moreover $box(\emptyset) = \emptyset$ and for each $\delta \subseteq V_n$: $box(\hat{\exists}_\Delta c) = \hat{\exists}_\Delta box(c)$ (i.e. $\lambda x.box(x)$ is weakly compatible).

The universe of abstract constraints $box(\mathcal{P})$ contains hypercubes and hyperplanes as constraints. Thus, future redundant constraints can be handled in the simpler non-distributive abstract constraint system $box(\mathcal{R}_n)$. Since $box(\mathcal{R}_n)$ is not Noetherian, termination conditions, such as the widening/narrowing techniques proposed in [12], have to be applied in the abstract fixpoint evaluations.

Better results can be obtained by keeping separated the answers for each clause in the program, with a less abstract semantic construction.

Recently, several studies have been devoted to "implement" interval arithmetics in the constraint logic programming paradigm. In [2] *bounding box* spatial approximations in constraint logic programs over finite domains are specified as an instance of our framework. In [27], the use of intervals has been presented to absorbe floating-point errors in $CLP(\mathcal{R})$ computations. They present an implementation based on a meta-interpreter executed by an existing $CLP(\mathcal{R})$ system [24]. Both of these approaches can be used for future redundant constraint detection. ◇

5.1 Linear Relationships Analysis

In this section we study the applicability of non-distributive constraint systems by modifying the ground dependency analysis technique to cope with linear relationships among predicate's arguments in $CLP(\mathcal{H})$.

A number of data-flow analyses in imperative languages are included in the determination of linear relations among variables, like compile-time overflow, integer subrange and array bound checking [15]. A very useful analysis on the relationships among variables of a program can be specified in our framework by *linear relationships analysis* [15,26,18,34], which provides useful information for proving termination, compile-time overflow, mutual exclusion, program debugging *etc.* The problem of discovering linear equality relations by abstract interpretation in logic programs has been studied in [34].

The automatic derivation technique in [34] for linear size relations among variables in logic programs can be suitably specified as a constraint computation. In the following we will show how this technique can be viewed as an instance of our framework, thus making explicit the strong relation between automatic detection of linear relationships among variables and $CLP(\mathcal{R})$ computations.

Let $\tau_{(\Sigma,V)}$ be defined as in Example 3.1, over a finite set of variables V. Let $|..|_\zeta$ be a (semi-linear) norm ([7]) on the term system $\tau_{(\Sigma,V)}$[7]. We define a term system τ_{Exp} of linear expressions where terms are first order terms in the language $\{+,0,1,V\}$ (i.e. terms in $\tau_{(\{+,0,1\},V)}$). Substitutions are performed as standard substitutions. In the following we represent the term $X+X+Y+1+1+1$ as $2X+Y+3$. τ_{Exp} is a term system.

The mapping $Exp_\zeta : \tau_{(\Sigma,V)} \rightarrow \tau_{Exp}$ associates a linear expression with each terms in $\tau_{(\Sigma,V)}$, as follows [34]:

[7] A norm $|\cdot|$ is said to be *semilinear* if it is of the form

$$|t| = \begin{cases} 0 & \text{if } t \text{ is a variable} \\ c + \sum_{i \in W} |t_i| & \text{if } t = f(t_1,\ldots,t_n), \text{ where } c \geq 0 \text{ and } W \subseteq \{1,\ldots,n\}. \end{cases}$$

$$Exp_\zeta(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ c_0 + \sum_{f \in F_t} Exp_\zeta(f(t)) & \text{otherwise} \end{cases}$$

Example 5.2 With *length* and *size* norms:

$|t|_{length} = 0$ if t is a variable,
$|t|_{length} = 0$ if $t = []$,
$|t|_{length} = 1 + |tail|_{length}$ if $t = [h|tail]$,
(this norm measures the length of a list)

$|t|_{size} = 1$ if t is a variable or a constant,
$|t|_{size} = 1 + |t_1|_{size}, ..., |t_n|_{size}$ if $t = f(t_1, ..., t_n)$,
(this norm measures the size of a term as the size of its subterms)

we have: $Exp_{length}([X[a|Z]]) = 1 + 1 + Z$ and $Exp_{size}([X[a|Z]]) = 1 + X + 1 + Z$ respectively.
◇

A constraint system of *affine relationships* (i.e. linear equalities of the form $c_0 = c_1 X_1 + ... + c_n X_n$) can be defined by specifying intersection, disjunction and cylindrification (variable restriction) as given in [34]. Intuitively, an *affine subspace* is a point, line, plane, *etc.*, possibly not including the origin. A *linear* subspace is an affine subspace containing the origin. Recall that an *affine transformation* $T : \Re^n \rightarrow \Re^m$ maps affine subspaces to affine subspaces; its kernel is the set of elements mapped to the origin, and is itself an affine subspace. Linear relations can be represented as n-tuples of real numbers (geometrically as sets of points in a n-dimensional space). These sets are approximated by *affine subspaces* or *linear varieties* [26,34]. A scheme for the finite representation of these (possibly infinite) spaces is provided by representing the space as the *kernel* of an *affine transformation* from $\Re^n$ to $\Re^m$ for appropriate m [26]. Affine transformations from $\Re^n$ to $\Re^m$ can be represented as an $m \times n$ matrix A together with an $m \times 1$ column matrix (vector) c. The corresponding transformation maps $x \in \Re^n$ to $Ax - c \in \Re^m$. The affine subspace can be found by solving the non-homogeneous system $A\ X\ =\ c$. Several different matrix-vector pairs may represent the same set of relationships. Elementary linear algebra fortunately provides us a "canonical form" for this problem. This canonical form can be obtained by reducing the augmented matrix $[A|c]$ in a *row-echelon form* [8]. Standard algorithms can be used to reduce any matrix in row-echelon form.

Consider the domain of affine subspaces $\mathcal{K}$ on a fixed n-dimensional space and the following basic operations, as given in [34]:

intersection ($\cap$): The *intersection* of two affine subspaces $[A_1|c_1]$ and $[A_2|c_2]$ is still an affine subspace. Such an intersection can be obtained by reducing the augmented matrix $\begin{bmatrix} A_1|c_1 \\ A_2|c_2 \end{bmatrix}$ to a row-echelon form. If the two affine subspaces have different dimension: m and $m + k$, we extend the one of lower dimension m to $m + k$ by adding k columns of 0's to the matrix and k rows of 0's to the corresponding vector.

[8] A matrix A is in row-echelon form iff every row has at least one non-zero entry, the first non-zero entry of each row is 1, for any row i_0 if j_0 is the first column with a non-zero entry of the row, then for all $i > i_0$, $j \leq j_0$: $A_{i,j} = 0$ and for all $i < i_0$, $A_{i,j_0} = 0$ [26].

union ($\hat{\oplus}$): The *union* of two affine subspaces is not, in general, an affine subspace. We consider instead the smallest affine subspace $[A|c]$ containing $[A_1|c_1]$ and $[A_2|c_2]$, namely if $[A_1|c_1]$, $[A_2|c_2]$ and $[A|c]$ specify linear transformations T_1, T_2 and T then $kernel(T)$ is isomorphic to $kernel(T_1) + kernel(T_2)$. In [26] an efficient algorithm to compute linear disjunctions has been introduced. Examples are shown in [34].

cylindrification ($\hat{\exists}$): The *variable restriction* operation is performed by cylindrification parallel to an axis. By definition, the cylindrification of an affine subspace is still an affine subspace. In [34] the cylindrification operation is defined as a matrix transformation.

substitution ($\hat{\delta}$): Let S be an affine subspace and $x \in V$, $t \in \tau_{Exp}$. Substitution of x with t in S is defined as the affine subspace $\hat{\exists}_{\{x\}}([x = t] \cap S)$.

Variables are assumed to be finite [9]: $V = V_n = \{x_1, ..., x_n\}$. If the relations are contradictory, then the subspace is the empty set $\emptyset$ (it cannot be represented as a pair matrix-vector). If there are no affine relations, the corresponding subspace is the entire space $\Re^n$. Diagonal elements are (single) equations on the term system τ_{Exp}. In the following, for each equation $t_1 = t_2$, we denote by $[t_1 = t_2] \subseteq \Re^n$ the corresponding affine subspace. As before, this notation simplifies somewhat the presentation.

Proposition 5.2
$(\mathcal{K}, \cap, \hat{\oplus}, \Re^n, \emptyset, \hat{\exists}_\Delta, \hat{\delta}^t_x, [t = t'])_{\{x\},\Delta\subseteq V_n; t,t'\in\tau_{Exp}}$ *is a non-distributive,* $\cap$*-idempotent and commutative constraint system, where* $\Re^n$ *is annihilator for* $\hat{\oplus}$.

As pointed out in [26], there are no infinitely ascending chains of free-variable bounded constraints (i.e. bounded dimension affine spaces), otherwise in any properly ascending chain of subspaces: $U_1 \hat{\trianglelefteq} U_2 \hat{\trianglelefteq} ...$ the subspaces U_i must have a dimension of at least one greater than U_{i-1}. The resulting constraint system is then Noetherian.

A linear equation is associated with each equation on terms. The following example shows the length relationships among the arguments of the **append** predicate. The solution can be obtained in a $\Re^3$ dimension space.

Example 5.3 Consider the logic program defining the predicate *append* in Example 4.1, together with the semilinear norm *length*. The corresponding abstract program is:

$$append(x_1, x_2, x_3) \;:-\; x_1 = 0, x_2 = x_3.$$
$$append(x_1, x_2, x_3) \;:-\; x_1 = 1 + y, x_3 = 1 + z \;\Box\; append(y, x_2, z).$$

The abstract semantics is:

$$T_P^{\mathcal{K}} \uparrow 0(\emptyset) = \emptyset$$
$$T_P^{\mathcal{K}} \uparrow 1(\emptyset) = \{append(x_1, x_2, x_3) \;:-\; x_1 = 0, x_2 = x_3\} \qquad [\; c_1 \;]$$

[9] As we are interested in relations (those defined in the program) having finite arity, we can always represent any answer constraint as a constraint on the finite dimensional space of its free variables. Moreover, the use of a bottom-up semantic construction does not require any infinite set of variables for renamings.

$$\begin{aligned} \mathcal{T}_P^{\mathcal{K}} \uparrow 2(\emptyset) &= \left\{ append(x_1, x_2, x_3) \ :- \ \left(\begin{array}{c} x_1 = 0, x_2 = x_3 \\ \hat{\oplus} \\ x_1 = 1, x_3 = 1 + x_2 \end{array} \right) \right\} && [\ c_1 \hat{\oplus} c_2\ ] \\ &= \{ append(x_1, x_2, x_3) \ :- \ x_1 + x_2 = x_3 \} && \text{(fixpoint)} \end{aligned}$$

Let us denote A_1 and A_2 the augmented matrices associated with the constraints c_1 and c_2 respectively (on the $3 + 1$-dimensioned space x_1, x_2, x_3, x_4). Applying the algorithm in [26] we have:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 1 \end{bmatrix} \quad \Rightarrow \quad A_1 \hat{\oplus} A_2 = \begin{bmatrix} 1 & 1 & -1 & 0 \end{bmatrix}.$$

The affine subspace $x_1 + x_2 = x_3$ specifies the affine relationship among the length of the arguments of the predicate *append* in the expected way. ◇

6 Related Work

Abstract interpretation of constraint logic programs is considered by Marriott and Søndergaard [29]. Their treatment is based on abstracting a denotational semantics for constraint logic programs. A meta-language based on the typed λ-calculus is used to specify the semantics of logic languages in a denotational style, and both the standard and non-standard semantics are viewed as instances of the meta-language specification. In our case, instead of defining a meta-language for data-flow analysis, we consider the constraint specification on which the CLP paradigm is defined. Non-standard semantics for a given constraint-based program can thus be obtained by appropriately modifying the underlying constraint system. In this way, data-flow analyses of logic-based languages can be specified as a standard constraint computation. No difference is introduced between the concrete programming language and the abstract one. They both derive from the same general specification of the CLP paradigm.

A related approach is also considered by Codognet and Filè, who give an algebraic definition of constraint systems and consider abstract interpretation of constraint logic programs [9]. However, the algebraic structure considered by these authors is very different: only $\otimes$-composition is considered. The notion of "computation system" is introduced but the underlying structure is not provided with a join operator. Because of this construction, mainly based on a generalization of the top-down SLD semantics, a loop-checker consisting in a "tabled" interpreter is needed. In our framework, by contrast, extraneous devices such as loop checking and tabulation are not considered. Instead, finiteness is treated simply as a property of the constraint system, expressed in terms of $\trianglelefteq$-chains. This allows non-standard computations to be specified as standard CLP computations over an appropriate (abstract) constraint system.

7 Conclusions

Weaker constraint systems can be considered, where for example distributivity does not hold. The distributivity restriction is not applicable to a wide class of static analysis problems including linear relationships, as shown in Section 5.1, and *range variable analysis*, based on an abstract lattice of intervals specifying the range of program variables [2]. Non-distributive constraint systems can be studied as a more general framework for constraint-based program analysis. A classification of the different constraint systems useful in data-flow analysis can be based on the set of properties they hold. A comparison with our framework can be useful to systematically derive those properties of the semantic construction that may be affected by a different constraint system definition.

Another aspect of the semantic construction is the use of variable hiding operators (such as cylindrifications) in the T_P definition. Technically, this allows the use of only finite sets of variables on which to perform renamings; thus simplifying the construction of finite upper approximations to the semantics, such as in the case of linear relationships analysis, where the finiteness is strongly related with the (finite) dimension of the space of solutions.

Acknowledgments

The stimulating discussions with Roberto Bagnara, Roberto Barbuti, Suzanne Dietrich, Maurizio Gabbrielli, Michael Maher, Nino Salibra, Gert Smolka, and David S. Warren are gratefully acknowledged.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley Publishing Company, 1974.

[2] R. Bagnara, R. Giacobazzi, and G. Levi. Static Analysis of CLP Programs over Numeric Domains. In *Actes Workshop on Static Analysis, WSA'92*, number 81-82 in Bigre, pages 43–50, 1992.

[3] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-flow Analysis. In *Proc of Ninth IEEE Conference on AI Applications*, pages 270–276. IEEE Computer Society Press, 1993.

[4] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[5] R. Barbuti and A. Martelli. A Structured Approach to Semantics Correctness. *Science of Computer Programming*, 3:279–311, 1983.

[6] G. Birkhoff. Lattice Theory. In *AMS Colloquium Publication, third ed.*, 1967.

[7] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky, and T.S.E. Maibaum, editors, *Proc. TAPSOFT'91*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, Berlin, 1991.

[8] J. Cirulis. An Algebraization of First Order Logic with Terms. *Colloquia Mathematica Societatis Jànos Bolyai*, 54:125–146, 1991.

[9] P. Codognet and G. Filè. Computations, Abstractions and Constraints. Technical Report 13, Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy, 1991.

[10] H. Comon and P. Lescanne. Equational Problems and Disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[11] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[13] P. Cousot and R. Cousot. A constructive characterization of the lattices of all retracts, pre-closure, quasi-closure and closure operators on a complete lattice. *Portugaliæ Mathematica*, 38(2):185–198, 1979.

[14] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[15] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. Fifth ACM Symp. Principles of Programming Languages*, pages 84–96, 1978.

[16] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238- 252. The MIT Press, Cambridge, Mass., 1991.

[17] M. Gabbrielli and G. Levi. A solved form algorithm for ask and tell Herbrand constraints. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1991.

[18] A. Van Gelder. Deriving Constraints Among Argument Sizes in Logic Programs. In *Proc. of the eleventh ACM Conference on Principles of Database Systems*, pages 47–60. ACM, 1990.

[19] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 581–591, 1992.

[20] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971.

[21] M. Hermenegildo, R. Warren, and S.K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

[22] Y.E. Ioannidis and E. Wong. An Algebraic Approach to Recursive Inference. In L. Kerschberg, editor, *Proc. First Int. Conf. Expert Database Systems - Charleston SC*, pages 295–309, 1987.

[23] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[24] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[25] N. Jørgensen, K. Marriot, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\mathcal{R}$). In *Proc. 1991 Int'l Symposium on Logic Programming*, pages 420–434, 1991.

[26] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[27] J. H. M. Lee and M. H. van Emden. Adapting CLP($\mathcal{R}$) to Floating-Point Arithmetic. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 996–1003, 1992.

[28] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[29] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 531–547. The MIT Press, Cambridge, Mass., 1990.

[30] Oystein Ore. Combinations of Closure Relations. *Annals of Mathematics*, 44(3):514–533, 1943.

[31] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, pages 333–353. ACM, 1991.

[32] Jichang Tan and I-Peng Lin. Compiling Dataflow Analysis of Logic Programs. In *ACM Programming Language Design and Implementation*, volume 27 of *SIGPLAN Notices*, pages 106–115. ACM Press, 1992.

[33] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[34] K. Verschaetse and D. De Schreye. Derivation of Linear Size Relations by abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 296–310. Springer-Verlag, Berlin, 1992.

Communications

AMAST'93

Third International Conference
on
Algebraic Methodology and Software Technology

University of Twente
The Netherlands

Participants' Proceedings

Dimension-Complemented Lambda Abstraction Algebras

Don Pigozzi and Antonino Salibra
Iowa State University and University of Bari

The untyped lambda calculus is formalized as a theory of equations, but it is not an equational theory in the usual algebraic sense because the equations, unlike the associative and commutative laws for example, are not always preserved when arbitrary terms are substituted for variables. Consequently the general methods that have been developed in universal algebra and category theory, for defining the semantics of an arbitrary algebraic theory for example, are not directly applicable. There have been several attempts to reformulate the lambda calculus as a purely algebraic theory. The earliest and best known, although apparently not motivated by these considerations, is the combinatory logic of Curry. More recently, several purely algebraic theories of the lambda calculus within the context of category theory have been developed: Obtułowicz and Wieger [9] via the algebraic theories of Lawvere; Adachi [1] via monads; Curien [3] via categorical combinators.

In [10] we proposed an alternative approach in the context of universal algebra. We introduced the notion of a *lambda abstraction algebra* (LAA for short), which is intended to provide a purely algebraic theory of the lambda calculus in the same way Boolean algebras constitute an algebraic theory of classical propositional logic and, more to the point, cylindric and polyadic Boolean algebras an algebraic theory of first-order predicate logic. In all algebraic theories of the lambda calculus the role of the variables is suppressed to varying degrees and the notion of substituting terms for the free variables of a term is abstracted. In LAA's this is effected by "inverting" (β)-conversion to obtain a definition of substitution in terms of the primitive notions of application and lambda abstraction.

The natural models of lambda abstraction theory are algebras of functions of possibly infinite arity, while models of the lambda calculus consist exclusively of unary functions. LAA's of functions of finite arity can be reduced to models of the lambda calculus by the well known method of Schönfinkel and Curry, but this is not possible in general. Consequently, there are functional LAA's with elements that cannot be represented by any term of the lamba calculus that is constructed from lambda variables and constants denoting the elements of some combinatory algebra. The dimension-complemented LAA's are the widest subclass of such algebras that are known to have a natural intrinsic characterization. In the present paper we prove that every dimension-complemented LAA is isomorphic to a point-relativized functional LAA.

The two primitive notions of the lambda calculus are *application* of a function to its argument (expressed as the juxtaposition of terms) and *lambda (functional) abstraction*, the process of forming a function from the "rule" that defines it. The connection between them is formalized in (β)-*conversion*: $(\lambda x.t)s = t[s/x]$. Here t and s are terms and $t[s/x]$ is the result of substituting s for all free occurrences of x in t, with the restriction that s must be "free for x in t".

A *lambda abstraction algebra* is an algebra of the form

$$\mathbf{A} = \langle A, \cdot, \lambda x_1, \lambda x_2, \ldots, x_1, x_2, \ldots \rangle,$$

where A is a nonempty set, $\cdot$ is a binary operation (corresponding to application), $\lambda x_1, \lambda x_2, \ldots$ is an infinite system of unary operations on A, and $x_1, x_2, \ldots$ a corresponding system of distinguished elements of A called *lambda variables*. Substitution is abstracted as a system of term-defined, binary operations $-[-/x_i]$ on A. The algebraic reformulation of (β)-conversion becomes the definition of abstract substitution:

$$b[a/x_i] ::= (\lambda x_i(a)) \cdot b, \qquad \text{for all } a, b \in A.$$

An element a of a LAA is said to be *algebraically dependent* on x_i if $a[x_j/x_i] \neq a$ for some $j \neq i$. A LAA is *locally finite-dimensional* if every element algebraically depends on only a finite number of x_i; it is *dimension-complemented* if, for each element a, there is at least one x_i on which a fails to depend. ¿From the axioms of LAA's given below it can be proved that a is in fact independent of an infinite number of x_i.

The equational axioms of LAA's reflect (α)-conversion and Curry's recursive axiomatization of substitution in the lambda calculus. They take the following form where Δa is the set of all x_i such that a is algebraically dependent on x_i.

$$(\beta_1)\ x_i[a/x_i] = a; \qquad (\beta_2)\ x_j[a/x_i] = x_j,\ j \neq i; \qquad (\beta_3)\ a[x_i/x_i] = a;$$

$$(\beta_4)\ \lambda x_i(b)[a/x_i] = \lambda x_i(b); \qquad (\beta_5)\ (b \cdot c)[a/x_i] = b[a/x_i] \cdot c[a/x_i];$$

$$(\beta_6)\ x_j \notin \Delta a \Rightarrow \lambda x_j(b)[a/x_i] = \lambda x_j(b[a/x_i]),\ j \neq i;$$

$$(\alpha)\ x_j \notin \Delta a \Rightarrow \lambda x_i(a) = \lambda x_j(a[x_j/x_i]).$$

Axioms (β_6) and (α) can be replaced by identities, so the class of LAA's forms a variety. The basic theory of LAA is developed in [10]. A closely related notion, lambda term systems, has recently been introduced by Diskin [4].

Theorem 1 *Let* $\mathbf{A}$ *be a dimension-complemented* LAA. *Then* $\Delta a_1 \cup \ldots \cup \Delta a_n$ *is coinfinite for any finite set* $a_1, \ldots, a_n$ *of elements of* $\mathbf{A}$.

The "intended" models of the theory are the *functional lambda abstraction algebras*.

Let $\mathbf{V} = \langle V, \cdot^{\mathbf{V}}, \lambda^{\mathbf{V}} \rangle$ be a structure where V is a nonempty set, $\cdot^{\mathbf{V}}$ is a binary operation on V, and $\lambda^{\mathbf{V}} : V^V \circ\!\!\to V$ is a partial function assigning elements of V to certain functions from V into itself. $\mathbf{V}$ is called a *functional domain* if, for every f in the domain of $\lambda^{\mathbf{V}}$, $f(v) = (\lambda^{\mathbf{V}}(f)) \cdot^{\mathbf{V}} v$, for all $v \in V$.

Let $\mathbf{V} = \langle V, \cdot^{\mathbf{V}}, \lambda^{\mathbf{V}} \rangle$ be a functional domain and let $V_\omega = \{ f : f : V^\omega \circ\!\!\to V \}$, where $\omega = \{1, 2, 3, \ldots\}$. By the *$\omega$-coordinatization* of $\mathbf{V}$ we mean the algebra

$$\mathbf{V}_\omega = \langle V_\omega, \cdot^{\mathbf{V}_\omega}, \lambda x_1{}^{\mathbf{V}_\omega}, \lambda x_2{}^{\mathbf{V}_\omega}, \ldots, x_1^{\mathbf{V}_\omega}, x_2^{\mathbf{V}_\omega}, \ldots \rangle,$$

where $\cdot^{\mathbf{V}_\omega}$, $\lambda x_i{}^{\mathbf{V}_\omega}$, and $x_i^{\mathbf{V}_\omega}$ are defined as follows: (for all $p \in V^\omega$, $v \in V$, and $p(v/i) \in V^\omega$ is defined as follows: $p(v/i)_j = v$ if $j = i$; $p(v/i)_j = p_j$ otherwise).

- $(a \cdot^{\mathbf{V}_\omega} b)(p) = a(p) \cdot^{\mathbf{V}} b(p)$, provided $a(p)$ and $b(p)$ are both defined; otherwise $(a \cdot^{\mathbf{V}_\omega} b)(p)$ is undefined.

- $\lambda x_i{}^{\mathbf{V}_\omega}(a)(p) = \lambda^{\mathbf{V}}(\langle a(p(v/i)) : v \in V \rangle)$, provided $\langle a(p(v/i)) : v \in V \rangle$ is in the domain of $\lambda^{\mathbf{V}}$ (note this implies $a(p(v/i))$ is defined for all $v \in V$); otherwise $\lambda x_i{}^{\mathbf{V}_\omega}(a)(p)$ is undefined.

- $x_i^{\mathbf{V}_\omega}(p) = p_i$.

A subalgebra **A** of total functions of $\mathbf{V}_\omega$, i.e., a subalgebra such that $a(p)$ is defined for all $a \in A$ and $p \in V^\omega$, is called a *functional lambda abstraction algebra*. Locally finite-dimensional functional LAA's are similar to the functional models of the lambda calculus developed in Krivine [6].

The locally finite-dimensional LAA's correspond most closely to the other algebraic models of the lambda calculus that have appeared in the literature, for instance the *term lambda algebras* ([7]) and *syntactical models* ([2]) of combinatory logic and the *Curry theories* of [9]. On the other hand *functional* LAA's correspond the *environment models* ([7]) and *lambda models* ([2]) of combinatory logic and the *functional Curry theories* in [9].

The following is the main result in [10].

Theorem 2 *Every locally finite-dimensional* LAA *is isomorphic to a functional* LAA *with the property that each function in the domain of the algebra depends on only a finite number of arguments.*

This theorem corresponds to the completeness theorem for the lambda calculus ([7]): every lambda theory consists of precisely the equations valid in some environment model. It is modeled on the representation theorem for locally finite-dimensional cylindric algebras ([5], Part II, Thm. 3.2.11(i)), which corresponds to the completeness theorem for first-order predicate logic (cf. the Forward of [5], Part I).

The representation of dimension-complemented LAA's requires a slightly more general notion of functional algebra

Let $\mathbf{V}$ be a functional domain. Let $\varepsilon \in V^\omega$ such that $\varepsilon(i) = x_i$ for all $i \in \omega$, and let V_ε^ω be the set of all $p \in V^\omega$ that differ from ε at only finitely many positions, i.e.,

$$V_\varepsilon^\omega = \{\, p \in V^\omega : |\{\, p_i \neq x_i \,\}| < \omega \,\}.$$

Let $V_{\omega,\varepsilon}$ be the set of all partial functions $f : V_\varepsilon^\omega \circ\!\!\rightarrow V$. The (I,ε)-*coordinatization* of V,

$$\mathbf{V}_{\omega,\varepsilon} = \langle V_{\omega,\varepsilon}, \cdot^{\mathbf{V}_{\omega,\varepsilon}}, \lambda x_1{}^{\mathbf{V}_{\omega,\varepsilon}}, \lambda x_2{}^{\mathbf{V}_{\omega,\varepsilon}}, \ldots, x_1^{\mathbf{V}_{\omega,\varepsilon}}, x_2^{\mathbf{V}_{\omega,\varepsilon}}, \ldots \rangle,$$

is defined just as $\mathbf{V}_\omega$ except that all functions are required to be in $V_{\omega,\varepsilon}$.

A subalgebra **A** of $\mathbf{V}_{\omega,\varepsilon}$ of total functions is called a *point-relativized functional lambda abstraction algebra.*

Theorem 3 *Every dimension-complemented lambda abstraction algebra is isomorphic to a point-relativized functional* LAA.

Outline of proof: Let **A** be an arbitrary LAA. The functional domain $\mathbf{V} = \langle V, \cdot^{\mathbf{V}}, \lambda^{\mathbf{V}} \rangle$ *associated with* **A** is defined as follows: $V = A$ and $\cdot^{\mathbf{V}} = \cdot^{\mathbf{A}}$. The domain of $\lambda^{\mathbf{V}} : V^V \circ\!\!\rightarrow V$ is $\{\, \langle a[v/x_i] : v \in V \rangle : a \in A \text{ and } i \in \omega \,\}$, and for each function in this set we define $\lambda^{\mathbf{V}}(\langle a[v/x_i] : v \in V \rangle) := \lambda x_i.a$. It can be shown that $\langle a[v/x_i] : v \in V \rangle = \langle b[v/x_j] : v \in V \rangle$ implies $\lambda x_i.a = \lambda x_j.b$. Thus $\lambda^{\mathbf{V}}$ is well defined. It is easily checked that **V** is a functional domain.

Let **A** be dimension-complemented and let **V** be its associated functional domain. For each $p \in V_{\omega,\varepsilon}$ there exist lambda variables $y_1, \ldots, y_n$ and elements $v_1, \ldots, v_n$ of V such that $p = \varepsilon(v_1/y_1, \ldots, v_n/y_n))$. Define a mapping $\Psi : A \to V_{\omega,\varepsilon}$ as follows: for all $a \in A$

$$\Psi(a)(\varepsilon(v_1/y_1, \ldots, v_n/y_n)) = a[z_1/y_1] \ldots [z_n/y_n][v_1/z_1] \ldots [v_n/z_n],$$

for all lambda variables $y_1, \ldots, y_n$ and all $v_1, \ldots, v_n \in V$ and any set of lambda variables $z_1, \ldots, z_n$ such that $a, y_1, \ldots, y_n, v_1, \ldots, v_n$ are all independent of each of the z_i. It can be shown that Ψ is well definded and an isomorphism between **A** and a total subalgebra of $\mathbf{V}_{\omega,\varepsilon}$.

It can be shown that the class of point-relativized functional LAA's (and their isomorphic images) form a variety. It coincides with the varieties generated by each of the classes of locally finite-dimensional, dimension-complemented, and functional LAA's. It is an open problem if functional LAA's form a variety and hence coincide (up to isomorphism) with point-relativized functional LAA's. Since the point-relativized functional LAA's form a variety they are axiomatized by some set of identities by Birkhoff's theorem. It is conjectured that they are finitely axiomatizable and, moreover, that the equational axioms for lambda algebras ([2], p. 94) together with those of LAA's are sufficient for this purpose. In contrast the representable cylindric algebras are not finitely axiomatizable.

Dimension-complemented LAA's have a direct analogue in the theory of cylindric algebras. Our representation theorem can be compared with the representation theory for dimension-complemented cylindric algebras; see [5], Part II, Thm. 3.2.11(ii). For a detailed survey of recent results in cylindric and related algebras see [8].

[1] T. Adachi, *A categorical characterization of lambda calculus models*, Research Report No. C-49, Dept. of Information Sciences, Tokyo Institute of Technology, January 1983.

[2] H.P. Barendregt, *The lambda calculus. Its syntax and semantics*, Revised edition, Studies in Logic and the Foundations of Mathematics, Vol. 103, North-Holland, Amsterdam, 1985.

[3] P.-L. Curien, *Categorical combinators, sequential algorithms and functional programming*, Pitman, 1986.

[4] Z.B. Diskin, *Lambda term systems*, preprint, 1990.

[5] L. Henkin, J.D. Monk and A. Tarski, *Cylindric algebras, Parts I and II*, North-Holland, Amsterdam, 1971, 1985.

[6] J.L. Krivine, *Lambda-Calcul, types et modeles*, Masson, Paris, 1990.

[7] A.R. Meyer, *What is a model of the lambda calculus?*, Inform. Control. 52(1982), 87–122.

[8] I. Németi, *Algebraizations of quantifier logics. An introductory overview*, Studia Logica, 50(1991), 485–569.

[9] A. Obtułowicz and A. Wiweger, *Categorical, functorial, and algebraic aspects of the type-free lambda calculus*, in: **Universal Algebra and Applications**, Banach Center Pub., vol. 9, Warsaw, 1982.

[10] D. Pigozzi and A. Salibra, *An introduction to lambda abstraction algebras*, to appear.

Parametrized Recursion Theory - A Tool for the Systematic Classification of Specification Methods

Till Mossakowski
University of Bremen
Department of Computer Science
P.O.Box 33 04 40
D-2800 Bremen 33
Fax: +49-421-218-4322
E-mail: e13p@alf.zfn.uni-bremen.de

Abstract

Parametrized recursion theory allows to characterize the power of parametrization in various specification methods. In particular, for the computation of the target algebra, the role of nondeterminism and the degree of availability of the parameter algebra can be studied.

Today, many different methods for the algebraic specification of abstract data types (ADTs) are proposed. They differ in their syntactical, semantical and categorical properties.
When you have a particular abstract data type in mind, which method should be used to specify it? If a certain method is not powerful enough, you have to choose a more general one. The other way round: if you use a too general method, then the available tools and proof techniques may become weaker. So it is very useful to know about which ADTs can be specified with the various methods at all.

1 Five Specification Methods

We compare five methods with increasing expressiveness:

1. total algebras with equations (see [EM85])
2. total algebras with equations and subsorting (see [Gog78])
3. total algebras with implications
4. total algebras with relations and implications (Horn Clause Theories, see [GM86, Pad88]) and finally
5. partial algebras with relations and implications built from existence-equations (algebraic systems in [Bur82]).

We use signatures $\Sigma = (S, OP, POP, REL)$ consisting of sort, total operation, partial operation and relation symbols. For simplicity, subsorting is coded by injection functions, so the second approach has special axioms *inj op* available, which specify an operation *op* to be injective. This can be expressed as $op(x) = op(y) \longrightarrow x = y$ in the third approach, so the approaches actually have increasing expressiveness.

Bergstra and Tucker [BT87] classify various specification methods with respect to recursion theoretic expressiveness of initial algebra semantics.
For designing modular specifications, parametrized specifications and data types are useful. We only consider parametrized data types (PADTs) which are specifiable with hidden sorts, operations and/or relations. That is, specifiable PADTs are composites of free and forgetful functors in the corresponding institution. In order to perform classifications with respect to PADTs, we first need a notion of computability over (parameter) algebras.

2 Computability over Abstract Algebras

In the literature. there are various approaches to define computability over an algebra.

Reichel [Rei87] defines T-algorithms for a theory T by using persistent extensions. This is no algorithmical or recursion theoretical concept, since it depends already on a particular specification method.
Kaphengst [Kap81] characterizes operations specifiable by free persistent extensions using effective numberings. But his characterization is not uniform: the extension of the parameter theory dependes on the parameter algebra.
Hupbach [Hup80] considers abstract implementations and characterizes specifiable functors by certain "uniform rules". This comes closer to our intention. The problem here is that "uniformity" is defined very technically during the proof of the characterization.
Bergstra and Klop [BK82] give an interesting characterization of specifiable functors with minimal total parameter algebras. For non-minimal algebras, they again have to incorporate some specification machinery and inital semantics in their notion of computable PADT (see [BK83]).

For parametrized recursion theory, we want a notion of *uniform algorithm* over abstract algebras, which is both algorithmic (hence does not rely on algebraic specification methods) *and* uniform, that is, independent of the particular representation of the parameter algebra (this corresponds to the "information hiding" principle). Moschovakis's prime and search computability [Mos69] fits into these requirements (see Ershov [Ers81] for an overview over the various approaches).

Natural numbers of ordinary recursion theory have to be replaced by another domain with pairing. Let $\Sigma = (S, OP, POP, REL)$ be a signature and A a Σ-algebra. The set $SExpr(A)$ of *S-expressions* over A is defined inductively: It contains nil, $atom\text{-}s(a)$ for $s \in S$, $a \in A_s$ and $cons(t, u)$ for $t, u \in SExpr(A)$. The set $SExpr$ is the subset of S-expressions containing no atoms. Like in LISP, we can consider natural numbers and lists of S-expressions again as S-expressions, and have $first$ and $rest$ as inverses of $cons$.

Moschovakis's approach also captures nondeterminism. He considers many-valued partial maps $\mathcal{F}_A: SExpr(A)^n \longrightarrow SExpr(A)$, such that for $\bar{t}$[1] $\in SExpr(A)^n$, the values z with $\mathcal{F}_A(\bar{t}) \rightarrow z$ form a (possibly empty) subset of $SExpr(A)$. $\mathcal{F}_A(\bar{t}) \simeq \mathcal{G}_A(\bar{t})$ means $\mathcal{F}_A(\bar{t}) \rightarrow u$ if and only if $\mathcal{G}_A(\bar{t}) \rightarrow u$.

Definition 2.1 (Moschovakis) Let $\Sigma = (S, OP, POP, REL)$ be a signature and $code: OP \dot{\cup} POP \dot{\cup} REL \longrightarrow \mathbb{N}$ some numbering. We define inductively the set of Σ-*algorithms* f as subset of $SExpr$.

	defintion scheme	S-expression f	
C0a.	$f(\bar{x}, y_1, \ldots, y_m) = op(\bar{x})$	$(0, n+m, code(op))$	$op: \bar{s} \longrightarrow s \in OP$
C0b.	$f(\bar{x}, y_1, \ldots, y_m) \simeq pop(\bar{x})$	$(0, n+m, code(pop))$	$op: \bar{s} \longrightarrow s \in POP$
C0c.	$f[R, y_1, \ldots, y_m] \simeq nil$	$(0, n+m, code(R))$	$R : \bar{s} \in REL$
C1.	$f(\bar{x}) \simeq nil$	$(1, n)$	
C2.	$f(y, \bar{x}) \simeq y$	$(2, n+1)$	
C3.	$f(t, u, \bar{x}) \simeq (t.u)$	$(3, n+2)$	
C4a.	$f(\bar{y}, x) \simeq first(x)$	$(4, n+1, 0)$	
C4b.	$f(\bar{y}, x) \simeq rest(x)$	$(4, n+1, 1)$	
C5.	$f(\bar{x}) \simeq g(h(\bar{x}), \bar{x})$	$(5, n, g, h)$	
C6.	$f(nil, \bar{x}) \simeq g(\bar{x})$	$(6, n+1, g, h_1, \ldots, h_m, k)$	$S = \{s_1, \ldots, s_m\}$
	$f(atom\text{-}s_i(y), \bar{x}) \simeq h_i(atom\text{-}s_i(y), \bar{x})$		$(i = 1, \ldots, m)$
	$f((t.u), \bar{x}) \simeq k(f(t, \bar{x}), f(u, \bar{x}), t, u, \bar{x})$		
C7.	$f(\bar{x}) \simeq g(x_{j+1}, x_1, \ldots, x_j, x_{j+2}, \ldots, x_n)$	$(7, n, j, g)$	
C8.	$f(e, \bar{x}, y_1, \ldots, y_m) \simeq \{e\}(\bar{x})$	$(8, n+m+1, n)$	
C9.	$f(\bar{x}) \simeq \nu y(g(y, \bar{x}) \rightarrow nil)$	$(9, n, g)$	□

Definition 2.2 A Σ-algorithm f has as semantics a familiy of many-valued partial maps $\{f\}^A: SExpr(A)^n \longrightarrow SExpr(A)$ indexed by Σ-algebras A. The semantical relation $\{f\}^A(\bar{t}) \rightarrow z$ is defined as the minimal relation satisfying the following conditions:

[1] We abbreviate $t_1, \ldots, t_n$ by $\bar{t}$, $x_1, \ldots, x_n$ by $\bar{x}$ and so on

Scheme		$f =$	
C0c.	$\overline{a} \in R_A \Rightarrow \{f\}^A(atom\text{-}\overline{s}_A(\overline{a}), u_1, \dots, u_m) \rightarrow nil$	$(0, n+m, code(R))$	$R : \overline{s} \in REL$
C5.	$\exists u\ (h(\overline{t}) \rightarrow u \wedge \{g\}^A(u, \overline{t}) \rightarrow v) \Rightarrow \{f\}^A(\overline{t}) \rightarrow v$	$(5, n, g, h)$	
C8.	$\{e\}^A(\overline{t}) \rightarrow v \Rightarrow \{f\}^A(e, \overline{t}, u_1, \dots, u_m) \rightarrow v$	$(8, n+m+1, n)$	
C9.	$\{g\}^A(u, \overline{t}) \rightarrow nil \Rightarrow \{f\}^A(\overline{t}) \rightarrow u$	$(9, n, g)$	

(For the other cases, the definition schemes are translated to semantical conditions similarly.) □

Schemes C0 to C7 allow to express *primitive recursiveness*, schemes C0 to C8 *prime computability* and schemes C0 to C9 *search computability* (with the ν-operator, an unordered, nondeterministic search is possible). Both prime and search computability reduce to partial recursiveness when Σ is empty.

Since the equality relation is not necessarily search computable, we have to add explicitly, if necessary, relation symbols $EQ\text{-}s : s\ s$ for $s \in S$ to parameter signatures Σ. The resulting signature is denoted by $EQ(\Sigma)$, and $EQ(A)$ interprets $EQ\text{-}s$ as equality on A_s.

Definition 2.3 We call a familiy $\mathcal{R} = (\mathcal{R}_A)_{A \in Alg(\Sigma)}$ of relations ($\mathcal{R}_A \subseteq SExpr(A)^n$) *primitively recursive (semi-search computable)*, if there is a primitively recursive (search computable) Σ-algorithm f with

$$\overline{t} \in \mathcal{R}_A \text{ iff } \{f\}^A(\overline{t}) \rightarrow nil$$

for all $A \in Alg(\Sigma), \overline{t} \in SExpr(A)^n$. We call $\mathcal{R}$ *primitively recursively enumerable*, if there is a primitively recursive Σ-algorithm f with

$$range(\{f\}^A) = \mathcal{R}_A$$

for all $A \in Alg(\Sigma)$. □

The computational model allows to make explicit the kind of nondeterminism and parallelism inherent in algebraic specification methods. Only semi-search computable families of relations are closed under unbounded search and existential quantification; and nondeterminism resp. OR-parallelism and full access to the parameter are available. Relations from primitively recursive and primitively recursively enumerable families are independent of the parameter operations and relations (see [Mos92])! That is, only the data sets can be used, but not equality or other relations and operations on the data. So primitively recursively enumerable families of relations are just closed under existential quantification. Primitive recursive families of relations are not even closed under existential quantification (this is well-known from ordinary recursion theory).

3 Computable PADTs

With the computational model of Moschovakis, we can generalize the notion of semi-computable algebra (see [BT87]) to the parametrized case.

Definition 3.1 Let $\Sigma \subseteq \Sigma 1$ be a parametrized signature ($\Sigma = (S, OP, POP, REL), \Sigma 1 = (S1, OP1, POP1, REL1)$). An algorithm p for a semi-search computable $(\Sigma, \Sigma 1)$-PADT is a quintuple $p = ((\chi_s)_{s \in S1}, (eq_s)_{s \in S1}, (\Phi_{op})_{op \in OP1}, (\Xi_{pop})_{pop \in POP1}, (\Psi_R)_{R \in REL1})$, where χ_s and Ψ_R (resp eq_s) are $EQ(\Sigma)$-algorithms for unary (resp. binary) semi-search computable families of relations, the Φ_{op} (resp. Ξ_{pop}) are $EQ(\Sigma)$-algorithms for primitively recursive (resp. search computable) families of maps of appropriate arity, such that for each Σ-algebra A

1. $(\{eq_s\}^{EQ(A)})_{s \in S1}$ has the formal properties of a closed congruence relation (see [Bur86]).
2. $Image(\{\Phi_{op}\}^{EQ(A)}) \subseteq \{\chi_s\}^{EQ(A)}$
3. For $s \in S$ we have: $\{atom\text{-}s(a) \mid a \in A_s\} \subseteq \{\chi_s\}^{EQ(A)}$
4. For $pop : \overline{s} \longrightarrow s \in POP$ and $\overline{a} \in$ dom pop_A we have $atom\text{-}s(pop_A(\overline{a}))\ \{eq_s\}^{EQ(A)}\ \{\Xi_{pop}\}^{EQ(A)}(atom\text{-}\overline{s}(\overline{a}))$, and analogously for $op \in OP$ and $R \in REL$. □

Definition 3.2 Let $\Sigma \subsetneq \Sigma 1$ be a parametrized signature and p an algorithm for a semi-search computable $(\Sigma, \Sigma 1)$-PADT. The semantics of p is the PADT $\{ p \} = (\eta, F)$ with $\eta \colon Id_{Alg(\Sigma)} \longrightarrow V_\Sigma \circ F$[2] and for each $A \in Alg(\Sigma)$

$$\begin{array}{ll}
\equiv_s := \{eq_s\}^{EQ(A)} & s \in S1 \\
(FA)_s := \{\chi_s\}^{EQ(A)} / \equiv_s & s \in S1 \\
op_{FA}([\bar{t}]_{\equiv_T}) \simeq [\{\Phi_{op}\}^{EQ(A)}(\bar{t})]_{\equiv,} & op \colon \bar{s} \longrightarrow s \in OP1 \\
pop_{FA}([\bar{t}]_{\equiv_T}) \simeq [\{\Xi_{pop}\}^{EQ(A)}(\bar{t})]_{\equiv,} & pop \colon \bar{s} \longrightarrow s \in POP1 \\
[\bar{t}]_{\equiv_T} \in R_{FA} \text{ iff } \{\Psi_R\}^{EQ(A)}(\bar{t}) - nil & R \in REL1 \\
\eta_{A,s}(a) := [atom\text{-}s(a)]_{\equiv,} & s \in S
\end{array}$$

□

4 The characterization

Theorem 4.1 Let $T \subsetneq T1$ be a parametrized theory in method i ($i = 1, \ldots, 5$) and (η, F) a persistent[3] PADT with $F \colon Alg(T) \longrightarrow Alg(T1)$, $\eta \colon Id_{Alg(T)} \longrightarrow V_T \circ F$. Then the follwoing are equivalent

(1) (η, F) is computable by an algorithm p according to row i in the table below.

(2) (η, F) is specifiable with method i. That is, there is a theory $T2$ with $T \subsetneq T1 \subsetneq T2$ such that for each T-algebra A,

$$V_{T1}\, F_{(T,T2)}\, A \cong F\, A$$[4]

and η_A is the parameter embedding of A into $F_{(T,T2)}\, A$.

Moreover, $T2$ can be computed effectively from p and vice versa (up to some emptyness problems, which are ignored here but can be solved, see [Mos92]).

Method	Recursion theory					Categorical	Example
	data	congruence	subsorts	relations	partial	property	PADT
	χ_s	on data	(range	Ψ_R	operations	of model	separating
		eq_s	Φ_{inj})		Ξ_{pop}	categories	the methods
1	pr.	p.e.	–	–	–	equivalences (see [MR77]) have quotients	lists, trees etc. over some data
2	pr.	p.e.	p.e.	–	–	coequalizers commute with subobjects	factorization over the image of some function
3	pr.	s.c.c.	p.e.	–	–	regular epis are pullback stable	making some Abelian group torsion free
4	pr.	s.c.c.	p.e.	s.c.c.	–	(reg epi,mono)-factorizations exist	transitive closure of some relation
5	s.c.c	s.c.c.	s.c.c.	s.c.c.	s.c.	locally finitely presentable category	set of paths over some graph

In the table, a "pr." means primitive recursiveness, an "(s.)s.c." means (semi-)search computability and a "p.e." means primitively recursive enumerability.
The total operations Φ_{op} always can be chosen primitively recursive. □

[2] V_Σ yields the Σ-reduct of a $\Sigma 1$-algebra

[3] that is, η is a natural isomorphism

[4] $F_{(T,T2)}$ is the free construction corresponding to the parametrized theory $T \subsetneq T2$

Interestingly, many differences shown in the table vanish in the unparametrized case. For example, both primitively recursive enumerability and semi-search computability then reduce to recursive enumerability. If uniformity considerations are ignored, the last four methods all have the same power (with initial semantics, and relations and (graphs of) partial functions possibly represented as subsorts), though the properties of the model categories differ. If you switch over to the parametrized case, then the recursion theoretical properties (of free constructions) get into a narrow correspondence with categorical properties (of loose semantics), especially concerning the behaviour of quotients. Thus, in a sense, parametrized recursion theory reconciles recursion theory with category theory.

The above results only hold for persistent parametrized data types. In the non-persistent case, the computational model has to be modified by some construction using inductive limits. This sheds some light on well-known difficulties with non-persistent parametrizations. See [Mos92].

References

[BK82] J.A. Bergstra and J.W. Klop. Algebraic specifications for parametrized data types with minimal parameter and target algebras. In *Proc ICALP 1982*, volume 140 of *SLNCS*, pages 23–34. Springer Verlag, 1982.

[BK83] J.A. Bergstra and J.W. Klop. Initial algebra specifications for parametrized data types. *Elektronische Informationsverarbeitung und Kybernetik*, 19:17–32, 1983.

[BT87] J.A. Bergstra and J.V. Tucker. Algebraic specifications of computable and semicomputable data types. *TCS*, 50:137–181, 1987.

[Bur82] P. Burmeister. Partial algebras — survey of a unifying approach towards a two-valued model theory for partial algebras. *Algebra Universalis*, 15:306–358, 1982.

[Bur86] P. Burmeister. *A model theoretic approach to partial algebras*. Akademie Verlag, Berlin, 1986.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer Verlag, Heidelberg, 1985.

[Ers81] A.P. Ershov. Abstract computability on algebraic structures. In A.P. Ershov and D.E. Knuth, editors, *Algorithms in Modern Mathematics and Computer Science*, volume 122 of *SLNCS*, pages 397–420. Springer Verlag, 1981.

[GM86] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming. Functions, Relations and Equations*, pages 295–363. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Gog78] J. A. Goguen. Order sorted algebras: Exceptions, error sorts, coercion and overloaded operators. Semantics and theory of computation report no. 14, University of California, Los Angeles., 1978.

[Hup80] U.L. Hupbach. Abstract implementation of abstract data types. In P. Dembiński, editor, *Proc. MFCS 1980*, volume 88 of *SLNCS*, pages 291–304. Springer Verlag, 1980.

[Kap81] H. Kaphengst. What is computable for abstract data types? In *Proc. FCT 1981*, volume 117 of *SLNCS*, pages 173–181. Springer Verlag, 1981.

[Mos69] Y.N. Moschovakis. Abstract first order computability I. *Transactions of the AMS*, 138:427–464, 1969.

[Mos92] T. Mossakowski. Spezifizierbarkeit und Berechenbarkeit parametrischer partieller Datentypen. Diploma thesis, Universität Bremen, 1992.

[MR77] M. Makkai and G.E. Reyes. *First Order Categorical Logic*. Springer Lecture Notes in Mathematics 611. 1977.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*. Springer Verlag, Heidelberg, 1988.

[Rei87] H. Reichel. *Initial Computability, Algebraic Specifications and Partial Algebras*. Oxford Science Publications, 1987.

Adding Algebraic Methods to Traditional Functional Languages by Using Reflection*

Tim Sheard
Oregon Graduate Institute of Science & Technology
sheard@cse.ogi.edu

Recent work by Malcom [7], Meijer, Fokkinga, and Paterson [8], and Cockett with the programming language **Charity** [1, 2] has suggested a high level of modularity and abstraction may be obtained by the use of generic control structures that capture patterns of recursion for a large class of algebraic types in a uniform way. This is important for several reasons.

- **Abstraction.** It allows the specification of algorithms independent of the type of data structures they are to operate on, since the control structure of the algorithm is generated for each datatype.
- **Genericity.** It allows the statement, proof, and use of type parametric theorems independant of any particular type.
- **Structure.** Functional programs are often the target of transformation and optimization. These techniques generally search for patterns of structure in programs to satisfy hypothesis of particular transformations. If structure is explicit, rather than implicit, the job of the transformation system is made easier.

Unfortunately it is hard to reap these benefits when using a traditional functional programming language as there is no mechanism for defining type parametric abstractions, which are the heart of many algebraic methods. This shortcomming can be overcome by the use of reflection in a typed language.

A programming language supports reflection if it has a distinguished class of values that correspond to syntactic fragments of the language and operations to manipulate these representation as data or programs, either by computing over them, evaluating them or injecting them into the value space. Typically these operations are called *reify* : *value* → *rep*, *reflect* : *rep* → *value*, and *eval* : *rep* → *rep*. We are going to concentrate on the uses of reflect.

Reflection is classified as either "compile time" or "run time" depending on when the semantic actions are expected to take place. Semantically, compile time reflection is the most straightforward since every compile time reflective program has the same meaning as a program that does not use reflection which is obtained by executing all of the reflection operations.

1 Type Parametric Combinators

Algebraic methods can be added to traditional functional languages by the disciplined use of compile-time reflection. Algebraic operators like *fold* can be created by computing over the representations of type declarations to build the representation of operators for these types, then reflecting over these representations to obtain the actual operators. For example, this could be done in the following way. Consider sum-of-products types defined by using recursive equations of the form:

$$T(\alpha_1,\ldots,\alpha_p) \;=\; C_1(t_{1,1},\ldots,t_{1,m_1}) \mid \cdots \mid C_n(t_{n,1},\ldots,t_{n,m_n})$$

*Tim Sheard is supported in part by a grant from OACIS and Tektronix

where $\alpha_1, \ldots, \alpha_p$ denote type variables, the C_i are names of value constructor functions, and $t_{i,j}$ are either type variables (in the set $\alpha_1, \ldots, \alpha_p$) or instances of sum-of-products types, including the type $T(\alpha_1, \ldots, \alpha_p)$ itself.

Functions manipulating values of these types will use a pattern of recursion related to the pattern of recursion in the type definitions. Algebraic methods often capture these patterns using the categorical notion of a functor. The functor, E^T,[5, 1, 14] defined below, is the morphism part of a categorical functor. There exists an E^T for each the type T. Category theorists would say that T is defined in terms of the fix point of E^T. Functional programmers are used to defining types by the use of recursive equations, so we follow this path.

Using E^T it is possible to describe the generalized *fold* (catamorphism [8]) operator for any simple sum-of-products type by defining a set of recursive equations, one for each constructor, C_i:

$$\text{fold}^T(\overline{h}) \circ C_i = h_i \circ E_i^T(id_1, \ldots, id_p, \text{fold}^T(\overline{h}))$$

where $\overline{h} = (h_1, \ldots, h_n)$ and for each index j, id_j is the identity function.

To make this definition precise we must provide a definition of E^T in terms of the data type equation defining T. The functor E^T is constructed from the n-fold sum of functors, E_i^T. Each, E_i^T is a $(p+1)$-adic functor [*] associated with the corresponding constructor, $C_i : (t_{i,1}, \ldots, t_{i,m_i}) \rightarrow T(\overline{\alpha})$.

$$E_i^T(\overline{f}, g_{Rec}) = \langle K^{T(\overline{\alpha})}[\overline{f}, g_{Rec}, t_{i,1}], \ldots, K^{T(\overline{\alpha})}[\overline{f}, g_{Rec}, t_{i,m_i}] \rangle$$

where $T(\overline{\alpha}) = T(\alpha_1, \ldots, \alpha_p)$, and $\overline{f} = f_{\alpha_1}, \ldots, f_{\alpha_p}$ and the notation $\langle h_1, \ldots, h_n \rangle$ represents a function with the property that $\langle h_1, \ldots, h_n \rangle (x_1, \ldots x_n) = (h_1\ x_1, \ldots, h_n\ x_n)$ and K is the type parametric combinator:

$$\begin{array}{lcl}
K^{T(\overline{\alpha})}[\overline{f}, g, \alpha_k] & = & f_{\alpha_k} \\
K^{T(\overline{\alpha})}[\overline{f}, g, T(\overline{\alpha})] & = & g \\
K^{T(\overline{\alpha})}[\overline{f}, g, S(t_1, \ldots, t_q)] & = & \text{map}^S\ (K^{T(\overline{\alpha})}[\overline{f}, g, t_1], \ldots, K^{T(\overline{\alpha})}[\overline{f}, g, t_q]) \\
K^{T(\overline{\alpha})}[\overline{f}, g, t_1 \times \ldots \times t_n] & = & \langle K^{T(\overline{\alpha})}[\overline{f}, g, t_1]\, x_1, \ldots, K^{T(\overline{\alpha})}[\overline{f}, g, t_n]\, x_n \rangle \\
K^{T(\overline{\alpha})}[\overline{f}, g, u \rightarrow v] & = & \lambda h.\ K^{T(\overline{\alpha})}[\overline{f}, g, v] \circ h \circ K^{T(\overline{\alpha})}[\overline{f}, g, u] \\
K^{T(\overline{\alpha})}[\overline{f}, g, ()] & = & id
\end{array}$$

We may also use E^S to generate the the morphism part of the categorical functor, often called the *map* for S:

$$(\text{map}^S(f_1, \ldots, f_p)) \circ C_i = C_i \circ (E_i^S(f_1, \ldots, f_p, \text{map}^S(f_1, \ldots, f_p)))$$

2 Compile-time Reflection

Language tools usually consist of an ***object language*** in which the programs which are being manipulated are expressed, and a ***meta language*** which is used to describe the manipulation. A compile time reflective language has features that allow it to be its own meta-language. We have built an implementation of compile-time reflection for a subset of ML we call CRML (Compile-time Reflective ML). In CRML the object language is "encoded" (represented) in an ML datatype. There is a datatype for each syntactic feature of ML. Object language manipulations are described by manipulations of this "representation" datatype. CRML contains syntactic sugar (object brackets << >>, and escape ') for constructing and pattern matching program representations which *mirror the corresponding actual programs*. Thus, meta programs manipulating object programs may either be expressed directly with the explicit constructors of the representation type or with this "object-language" extension to ML's syntax. Text within the object-language brackets (<< >>) is parsed but not compiled. Its representation is returned as the value. Meta-language expressions may be included in the object-language text by "escaping" them with a backquote character ('). Samples of this feature are illustrated in the table below.

[*] Where p is the number of universally quantified type variables in the left hand side of T's type equation.

Concrete syntax	Constructor based	Object bracket based
`x`	`Id "x"`	`<< x >>`
`f x`	`App(Id "f",Id "x")`	`<< f x >>`
	`App(g,y)`	`<< 'g 'y >>`
`(x,y)`	`Tuple [ Id "x", Id "y" ]`	`<< (x,y) >>`
	`Tuple [x, y]`	`<< 'x * 'y >>`

By using reflection, generic operators, such as *map* and *fold*, have straightforward implementations by computing over the representations of datatype declarations. In CRML a *template* defines a function which, when invoked, is mapped over all the constructors (and their corresponding types) of a datatype declaration, constructing the object language value for the representation of a function declaration. For example the template below defines a function `mapf` which generates the representation of a function declaration from a string (representing the name of a type constructor).

```
fun template mapf T =
    map f ((Ci of d -> r) xbar) = 'Ci ('(K r <<f>> <<map f>> d) 'xbar);
```

The expression in the constructor position of the function definition, `((Ci of d -> r) xbar)`, is treated as a pattern. Thus upon invocation of the template the variables in this pattern will be bound to object language values particular to each constructor. `Ci` is bound to an object language expression for the constructor function, `xbar` to an object language tuple expression (of the appropriate "shape" to be `Ci`'s argument), `d` to the object language type of `Ci`'s domain, and `r` to the object language type of `Ci`'s range (which is the type `T`).

The rest of the expression is taken literally to compute one of the equations defining a function, except that escaped expressions are evaluated at invocation time and "spliced" into the equation.

While an escape character inside object brackets or a template definition allows the results of meta computations to be "spliced" into object programs, an unbracketed, escaped expression is a simple interface to compile-time reflection. It indicates that the escaped expression should be evaluated (at compile-time) to compute the expression (or type, pattern, declaration, etc.) that replaces the escaped expression (much like macro expansion).

Thus, using the `mapf` meta program the program below calculates and defines the map for list.

```
val maplist = let '(mapf "list") in map end;
```

as if the user had typed the following instead:

```
val maplist = let fun map f [] = []
                  | map f (a1::a2) = Cons(f a1,map f a2)
              in  map end
```

3 Monadic Composition

We have used similar methods in automating the generation of polymorphic functions to realize the *monadic* structure of datatype declarations [6]. Moggi has shown that monads can be used to structure semantics [9]. Other researchers, including Wadler [13] and our group [6] have explored the use of monads to structure specifications and programs. Many algorithms may be expressed solely in terms of the monadic operations. When this can be done, changes to the details of the data type do not require changes in the specification of the algorithms. They also support a very powerful notion of composition that allows programs to be decomposed into more easily understood and maintained modules.

For example, let the type constructor *Maybe* be defined by $Maybe(x) \; = \; Nothing \mid Just(x)$. Spivey[11] has used this type to model exceptionai computations. *Maybe* has the structure of a monad[12, 6]. The ***binary product distribution*** for *Maybe*, with type $(Maybe(a) \times Maube(b)) \rightarrow Maybe(a \times b)$, can be defined as:

$$\tau_2^{Maybe}(x_1, x_2) \; = \; \{\, (a_1, a_2) \mid a_1 \leftarrow x_1;\; a_2 \leftarrow x_2 \}^{Maybe}$$

Using the usual translation[12] for monad comprehensions we get:

$$\tau_2^{Maybe}(x_1, x_2) = mult^{Maybe}\,(map^{Maybe}\,(\lambda\, a_1.(map^{Maybe}\,(\lambda\, a_2.(a_1 . a_2))\, x_2))\, x_1)$$

let the type $T(\alpha) = S(Maybe(\alpha))$, where S is any sum-of-products type. Then T has the structure of a monad[6]. The the distribution function $\pi^S_{Maybe} : S(Maybe(\alpha)) \rightarrow Maybe(S(\alpha))$ can be given in terms of the operator fold^S,

$$\pi^S_{Maybe}\, x = \text{fold}^S\,(f_1 \ldots f_n)\, x$$

where f_i is an accumulating function for each data constructor, $C_i : (\sigma_1 \ldots \sigma_{m_i}) \rightarrow S$. If C_i is a nullary constructor, C_i, then $f_i\,() = unit^{Maybe}\, C_i$. If C_i is not a nullary constructor, then the corresponding accumulating function, f_i, can be defined as

$$f_i = (map^{Maybe}\, C_i) \circ \tau^{Maybe}_{m_i} \circ H_i^S(unit^{Maybe}, id, id)$$

where H_i^S can be defined in a manner similiar to E_i^T as follows:

$$H_i^S(f_{non}, f_\alpha, f_{rec}) = \langle \dot{K}[\sigma_1]x_1, \ldots, \dot{K}[\sigma_{m_i}]x_{m_i}\rangle$$

and $\dot{K}$ is the type parametric combinator:

$$\begin{array}{lcl}
\dot{K}[t] & = & f_{non} \text{ when neither } \alpha \text{ nor } S(\alpha) \text{ occurs in t} \\
\dot{K}[\alpha] & = & f_\alpha \\
\dot{K}[S(\alpha_1, \ldots, \alpha_p)] & = & f_{Rec} \\
\dot{K}[U(t_1, \ldots, t_q)] & = & \text{map}^U\,(\dot{K}[t_1], \ldots, \dot{K}[t_q]) \\
\dot{K}[t_1 \times \ldots \times t_n] & = & \lambda(x_1, \ldots, x_n).(\dot{K}[t_1]\, x_1, \ldots, \dot{K}[t_n]\, x_n)
\end{array}$$

For example, the type composition distribution function, π^{List}_{Maybe}, is a function with type $List(Maybe(\alpha)) \rightarrow Maybe(List(\alpha))$, and can be defined as follows:

$$\pi^{List}_{Maybe}\, x = \text{fold}^{List}\,(f_{Nil}, f_{Cons})\, x$$

$$\text{where} \left\{ \begin{array}{lcl} f_{Cons}(x, xs) & = & \text{map}^{Maybe}\, Cons\,\,(\tau_2^{Maybe}(x, xs)) \\ f_{Nil}() & = & unit^{Maybe} Nil \;=\; Just(Nil) \end{array} \right.$$

This function, which can be generated for any datatype, S, allows us to *lift* a function, $f : \alpha \rightarrow Maybe(\beta)$ to a function, $g : S(\alpha) \rightarrow Maybe(S(\beta))$.

$$g = \pi^S_{Maybe} \circ (map^S\; f)$$

Using other type parametric combinators we have implemented algebraic generators for structural equality, and unification over data structures which represent abstract terms.

In addition we have defined an ***normalization algorithm*** [3, 4, 10] which automatically calculates improvements to programs whose only contol structures are folds. It reduces these programs to a canonical form. Based upon a generic ***promotion theorem*** [7, 8], the algorithm is facilitated by the explicit structure of fold programs rather than using an analysis phase to search for implicit structure. Canonical programs are minimal in the sense that they contain the fewest number of fold operations. Because of this property the improvement algorithm has important applications in program transformation, optimization, and theorem proving.

4 Conclusion

A compile-time reflective programming environment is an appropriate choice when computations over programs is necessary. Meta-programs can access the types of objects in the environment, retrieve representations of types or functions as data, generate representations of the derivative functions for types, or apply optimizations or transformations to functions, and then submit these representations to the compiler. This allows the incremental expansion of traditional functional languages to include algebraic methodologies based upon formal foundations in a straight forward manner.

References

[1] J. Cockett and D. Spencer. Strong Categorical Datatypes I. In R. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings, Vol. 13, pp 141-169. AMS, Montreal, 1992.

[2] J. Cockett and T. Fukushima. About Charity The University of Calgary, D·partment of Computer Science, Research Report No. 92/480/18. June 1992.

[3] L. Fegaras. *A Transformational Approach to Database System Implementation*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, February 1993. Also appeared as CMPSCI Technical Report 92-68.

[4] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In *Proceedings of the 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York*, pp 148-162. Springer-Verlag, June 1992.

[5] T. Hagino. *A Categorical Programming Language*. Ph.D. thesis, University of Edinburgh, 1987.

[6] James Hook, Richard Kieburtz, and Tim Sheard, Generating Programs by Reflection. Oregon Graduate Institute Technical Report 92-015, submitted to Journal of Functional Programming.

[7] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pp 335-347. Springer-Verlag, June 1989.

[8] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124-144, August 1991.

[9] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55 92, July 1991.

[10] T. Sheard. and L. Fegaras. A Fold for All Seasons. To appear: *Functional Programming Languages and Computer Architecture*, Copenhagen, June 1993.

[11] M. Spivey. A Functional Theory of Exceptions. In *Science of Computer Programming*, 14:25-42, 1990.

[12] P. Wadler. Comprehending Monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pp 61-78, 1990.

[13] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 1992.

[14] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118,127. Springer-Verlag, 1989.

A Coherent Type Inference System for a Concurrent, Functional and Imperative Programming Language *

D. Bolignano M. Debabi

Bull Corporate Research Center
Rue Jean Jaurès,
78340 Les Clayes-Sous-Bois
FRANCE

bolignano@frcl.bull.fr, debabi@frcl.bull.fr

ABSTRACT

The type and effect discipline is a recent framework for typing expressions in implicitly, strongly typed, polymorphic functional languages with imperative extensions. In addition to the automatic reconstruction of the principal types, this discipline computes the minimal side effects of expressions. The main objective of this work is to present a generalization of this discipline to concurrent, functional and imperative languages. Accordingly, we present an effect-based static semantics as well as an operational semantics for a language that unifies the previously mentioned computational styles. The proposed type system is applied to a concurrent ML-like language. It computes in addition to the type of expressions, their side and communication effects. Furthermore, an operational semantics of the language is presented. The latter is based on an extension of the Hennessy's operational semantics of the VPLA (Value-Passing Language with Assignment) language. That is why our dynamic semantics can be viewed as an extension of the work on CCS without τ's proposed initially by De Nicola and Hennessy. A significant goal of this paper is to prove that the static and the dynamic semantics are consistently related.

Keywords: Typing; Side and Communication Effects; Static and Dynamic Semantics; Concurrent, Functional and Imperative Programming; Process Algebra; Consistency of Typing.

1 Motivation and Background

The ultimate goal of this work is the definition of a wide-spectrum specification language that supports both data and concurrency descriptions. The design of this specification language, followed the same approach as the one used in the design of Extended ML [ST85]. The starting point is the design of an implicitly typed, polymorphic, concurrent and functional programming language. Axioms are then added in the signatures and structures as in Extended ML. The resulting specification language is thus highly expressive though it embodies a restricted number of concepts. More concisely our language can be viewed as a sugared version of typed λ-calculus that safely incorporates imperative and concurrent extensions.

The intent of this paper is to focus on the theoretic foundations of the underlying programming language. The latter unifies three computational paradigms which we refer to as concurrent, functional and imperative programming. A great deal of interest has been expressed in each of these programming styles and the underlying foundations have been deeply investigated, albeit generally separately.

Concurrency models have been a focus of interest for a great number of researchers. Accordingly, this gave rise to plenty of calculi and models. Prominent calculi are those that correspond to process algebra such as: CCS (Calculus for Communicating Systems) [Mil89] and CSP (Communicating Sequential Processes) [Hoa85] for which mathematically well-behaved models have been advanced. One can cite the failure-sets model of Brooks, Hoare and Roscoe [BHR84] or the acceptance-trees model of Hennessy [Hen85]. However, in spite of the large activity of the concurrency community, it remains that formalisms and techniques devised for concurrent and distributed systems are generally relevant to pure processes, in other words, they focus on control aspects rather than data aspects. Thus, in such frameworks, there is no data, no communication, no states,...etc. These simplifications are generally adopted in order to put the emphasis on the difficulties inherent to concurrent systems, for instance, nondeterminism, the semantics of combinators,...etc.

On the other hand, functional programming has been extensively studied. Consequently, many powerful, general-purpose programming languages emerged such as ML dialects. The latter rests on secure theoretical foundations that are exemplified by the large body of results on pure and typed λ-calculus. Generally, functional languages are endowed with imperative features for efficiency reasons. Also programming without such facilities becomes quickly tedious and cumbersome in many situations.

The language described here supports polymorphic types. It supports also both functional and process abstractions as in CML [Rep91] and FACILE [GMP89]: functions may be used to describe internal computations of concurrent processes. Functions, processes, references and communication channels are first-class values and thus can be passed along channels. Consequently, the mobility of these values is supported.

At the theoretical level, we will present the static semantics of this language as well as the dynamic semantics. The type inference system is based on an extension of the type and effect discipline: a new approach to implicit typing that can be viewed as an *extension* of the ML-style type discipline. In addition to that, as shown in [TJ92], effect-based type disciplines are more appropriate for integrating safely and efficiently functional and imperative programming. The dynamic semantics presented here is operational. It is based on an extension of VPLA (Value Passing Language with Assignment) of Hennessy [HI90, HI91]. Thus, the presented model can be viewed as a CCS without τ's version.

2 Related Work

Recently, some modern languages have been proposed that reconcile the functional, concurrent and imperative styles. For instance one can cite CML [Rep91], FACILE [GMP89] and LCS [Ber89]. All the three languages emerged from the idea of combining an SML-like language [MTH90] as a functional and imperative core, with a CCS or CSP-like process algebra for process abstraction. They support polymorphism, functional and process abstractions, dynamic behaviors and higher order objects.

The static semantics (typing semantics) in CML, FACILE and LCS rests on the type inference discipline. It is well known that this discipline, is problematic in the presence of non referentially transparent constructs. More precisely, the problem is relevant to type generalization in the presence of mutable data. Therefore, many extensions of the initial work of Milner [Mil78] have been proposed.

The classical way to deal with this issue, is the imperative type discipline [Tof87]. An extension of this approach has been used in the implementation of Standard ML of New Jersey. It is based on weak type variables: these type variables have an attached strength information, denoting the number of applications needed to get a non trivial effect. In [LW91], another method is proposed that consists in detecting some so called dangerous type variables (the ones occurring in the types of imperative objects), and labeling function types accordingly.

Later, in [TJ92], the type and effect discipline is introduced. The latter yields as a result of the static evaluation of an expression, not only its principal type, but also all the minimal side effects. A decidable and consistent typing system w.r.t. the operational semantics of the considered language, is advanced [TJ92]. Notice, that the inference typing system was devised for an ML-like language, of course with imperative constructs. It should be noted that the idea of considering the effects as part of the static evaluation of an expression, has been suggested in [Luc87] and adopted in the FX project [GJLS87, LG88].

As we pointed out before, one of the aims addressed here, is to propose a dynamic semantics for our language. Notice that elaborating a dynamic semantics for such languages is somewhat complicated. The reasons for this are that we have to deal with various aspects of the language (concurrent, functional and imperative). Another source of complication is the integration of all these aspects. Most of the dynamic semantics proposed for these languages (Concurrent ML-like) are operational. For instance, CML and FACILE are endowed with an operational semantics reported respectively in [BMT92, Rep91] and [GMP89]. Another description of FACILE semantics has been developed using the CHAMs [BB91] (CHemical Abstract Machines) framework [LT92]. In this paper, we present an operational semantics of our language that can be viewed as an extension of the VPLA operational semantics. Notice that a denotational model have been devised for our language. The model is briefly discussed in[BD93], its foundations are investigated in [BD92].

Our concern in this paper is:

- To propose a new inference typing system (implicit typing)that computes in addition to the principal types of expressions and their side effects, the minimal communication effects generated by the concurrent constructs.
- To propose an adequate operational semantics for our language (Concurrent ML-like).
- To prove that our typing system is consistent w.r.t. the static semantics. Notice that this issue is one of the most interesting results of this work.

3 Informal presentation

The syntactic constructions allowed in our language are close to those allowed in CML and FACILE. The set of expressions includes:

- Literals such as integers, booleans true and false, a distinguished value (), a constant skip which models an expression that immediately terminates successfully.

- Three binding operations that are the abstraction, the recursion and the let definition.
- Imperative aspects are supported through the notion of reference. Expressions of the form **ref** (E) stands for the allocation of a new reference and assigns to it the value obtained by evaluating the expression E. We will use the unary operator ! for dereferencing and the binary operator := for assignment.
- Expressions may communicate through channels. The expression **channel**() means allocate a new channel. The expression $E!E'$ means: evaluate E', evaluate E and send then the result of E' evaluation on the channel resulting from the evaluation of E. The whole expression evaluates then to (). The expression $E?$ evaluates to any value received on the channel resulting from the evaluation of E. Notice that the communications are synchronized as in CCS and CSP.
- Three concurrency combinators:

  $\_\sqcap\_$: Nondeterministic choice between two expressions (also called internal choice).

  $\_\Box\_$: External choice between two expressions.

  $\_\|\_$: Parallel composition of two expressions.
- A sequencing operator: $[\_;\_;]$.

More formally the BNF syntax of our language is:

$$\begin{aligned} E ::= {} & () \mid \mathit{true} \mid \mathit{false} \mid \mathit{Number}\ n \mid \mathit{ident}\ x \mid \mathbf{skip} \mid \\ & \lambda x \bullet E \mid E\ E \mid E \sqcap E \mid E \Box E \mid E \| E \mid E;E \mid \\ & \mathbf{ref}\ E \mid !E \mid E := E \mid \mathbf{channel}() \mid E? \mid E!E \mid \\ & \mathbf{if}\ E\ \mathbf{then}\ E\ \mathbf{else}\ E \mid \mathbf{let}\ x = E\ \mathbf{in}\ E \mid \\ & \mathbf{rec}\ x \bullet E \end{aligned}$$

In the following, we will use $\mathcal{P}_f$ to stand for the finite powerset, $A \overset{m}{\to} B$ for the set of all finite mappings (maps for short) from A to B.

4 Static semantics

As we pointed out before, we have adopted the type and effect discipline in order to give a static semantics to our language. This choice is motivated by the following reasons:

- As shown in [TJ92], the type and effect discipline is more appropriate than the other type systems [Tof87, LW91] in integrating efficiently functional and imperative programming. The reader should refer to [TJ92] for a full comparison of the type and effect discipline with the other approaches.
- A more efficient type generalization in let expressions by the use of the effect information and the observation criterion.
- One of the main and the most motivating reasons for us is purely technical and is relevant to the foundations of the denotational model. More accurately, CML-like languages in general and the language described here in particular are quite expressive. For instance higher order processes are allowed i.e processes are values and can be communicated along channels. Then if we attempt naively to construct the process domain, this will lead to reflexive domain definitions that have no solutions. In order to get round this difficulty, the technique considered in this work makes a dependence between the static and the dynamic semantics by typing the dynamic domains by the hierarchy laid down by the static domains. At this level we need to know exactly the type, the communication and the store effects of the language expressions. This issue is discussed in details in the next section.

The reader should notice that the type and effect discipline reported in [TJ92] does not support communication effects. Thus the work reported hereafter is an extension of this discipline. We define the following static domains:

- The domain of *Reference regions*: The notion of reference regions is introduced to abstract the memory locations. Every data structure corresponds to a region. Two values are in the same region if they may share some memory locations. The domain consists in the disjoint union of a countable set of constants and variables noted γ. We will use $\rho, \rho', \ldots$ to represent reference regions.
- The domain of *Reference effects*: Reference effects abstracts the memory side-effects. We define the following basic effects: $\emptyset$ for the absence of effect, ς for a reference effect variable, $init(\rho, \tau)$ for the reference allocation, $read(\rho)$ for reading in the region ρ and $write(\rho)$ for assignments of values to references in the region ρ. We introduce also a union operator $\cup$ for effects.

  $$\sigma ::= \emptyset | \varsigma | init(\rho, \tau) | read(\rho) | write(\rho) | \sigma \cup \sigma$$

  We will write $\sigma \sqsupseteq \sigma' \Leftrightarrow \exists \sigma'' \bullet \sigma = \sigma' \cup \sigma''$. Equality on reference effects is modulo ACI (Associativity, Commutativity and Idempotence) with $\emptyset$ as the neutral element.

Analogously, we introduce the following static domains:

- The domain of *Channel regions*: As with reference regions, channel regions are intended to abstract channels. Their domain consists in the disjoint union of a countable set of constants and variables noted δ. We will use $\chi, \chi', \ldots$ to represent values drawn from this domain.

- The domain of *Channel effects*: It is defined inductively by:

  $$\kappa ::= \emptyset|\eta|chan(\chi, \tau)|in(\chi)|out(\chi)|\kappa \cup \kappa$$

  We will use η to stand for a channel effect variable. The basic channel effect $chan(\chi, \tau)$ represents the creation of a channel of type τ in the channel region χ. $in(\chi)$ denotes the effect resulting from an input on a channel of the channel region χ while $out(\chi)$ denotes an output on the channel of the region χ. We will write $\kappa \sqsupseteq \kappa' \Leftrightarrow \exists \kappa'' \bullet \kappa = \kappa' \cup \kappa''$. Equality on effects is modulo ACI with $\emptyset$ as the neutral element.

- The domain of *types*: It is inductively defined by:

  $$\begin{aligned}\tau ::= &\ Unit|Bool|Int|\alpha| \\ &\ ref_\rho(\tau)|chan_\chi(\tau)|\tau \xrightarrow{\sigma,\kappa} \tau\end{aligned}$$

  $Unit$ is a type with only one element "()", α a type variable, $ref_\rho(\tau)$ is the type of references in the region ρ to values of type τ, $chan_\chi(\tau)$ is the type of channels in the communication region χ that are intended to be mediums for values of type τ, $\tau \xrightarrow{\sigma,\kappa} \tau'$ is the type of functions that take parameters of type τ to values of type τ' with a *latent* reference effect σ and a latent channel effect κ. We mean by latent effect, the effect generated when the corresponding expression is evaluated.

We also define type schemes of the form $\forall v_1, ..., v_n \bullet \tau$ where v_i can be type, reference region, channel region, reference effect and channel effect variable. A type τ' is an instance of $\forall v_1, ..., v_n \bullet \tau$ noted $\tau' \prec \forall v_1, ..., v_n \bullet \tau$, if there exists a substitution θ defined over $v_1, ..., v_n$ such that $\tau' = \theta\tau$.

Our static semantics contain sequents of the form:

$$\mathcal{E} \vdash E : \tau, \sigma, \kappa$$

which state that under some typing environment $\mathcal{E}$ the expression E has a type τ, a reference effect σ and a channel effect κ. Notice that type environments $\mathcal{E}$ map identifiers to type schemes.

5 Operational semantics

In this section we present the operational semantics. We will use the same style as [HI90][HI91]. For that, let us introduce first the notion of *computable values* of the language.

Definition 5.1 *The set $\mathcal{V}$ of computable values is defined as the least set which satisfies:*

- *$\mathcal{V}$ contains literals such as (),* true, false, *integers, or references, or channels.*
- *if $v, v' \in \mathcal{V}$, then $(v, v') \in \mathcal{V}$.*
- *If Γ is an environment, then the closure $[\![\lambda x \bullet E,\ \Gamma]\!] \in \mathcal{V}$.*

Let us denote by R the set of references and by K the set of channels. Now, we need to define the notion of *store*. The set of possible stores $Store$ is made of *store actions*. The latter stands for both the current associations of the references and values, and also for the different actions on the store (read, write operations and the channel creations). The formal definition is:

$$\begin{aligned}Store &= \mathcal{P}_f(Store\_Action) \\ Store\_Action &= \{init(r, v) \mid r \in R \text{ and } v \in \mathcal{V}\} \cup \\ &\quad \{read(r) \mid r \in R\} \cup \\ &\quad \{write(r) \mid r \in R\} \cup \\ &\quad \{chan(c) \mid c \in K\}\end{aligned}$$

The store action $init(r, v)$ means that the reference r is bounded to the value v. The store actions $read(r)$ and $write(r)$ model respectively a read and a write operation on the reference r. Finally, the store action $chan(c)$, corresponds to the creation of a channel c. We will write $s, s', ...$ to denote stores drawn from the set $Store$. We write s_r to denote the store s excluding store actions of the form $init(r, v)$. We say that s is included in s', or s' extends s, noted $s \subseteq s'$, if and only if there exists s'' such that $s' = s \cup s''$. We note $dom(s) = \{r \mid \exists v \bullet init(r, v) \in s\}$ the domain of store s.

We note $\mathcal{EV}$ the set of expressions and computable values. We will use $v, v', ...$ to represent values drawn from $\mathcal{V}$, $t, t', ...$ to represent values drawn from $\mathcal{EV}$ and E, $E_i, ...$ to represent expressions.

Our operational semantics is based on the evolution of special *configurations* defined hereafter. First, we distinguish the set of *basic* (initial) configurations:

Definition 5.2 *The set of basic configurations BC is defined as:*

$$BC = \{\langle t, s\rangle \mid t \in \mathcal{EV} \wedge \ s \text{ a store}\}$$

Definition 5.3 *The set of configurations, C, is defined as the least set, which satisfies:*

1. *$BC \subseteq C$*
2. *$\alpha \in C$ implies* ref *α, $\alpha?$, $!\alpha \in C$*
3. *$\alpha, \beta \in C$ implies α op β $\in C$ where: op = $\sqcap, \square$*
4. *$\alpha \in C$ implies $\alpha; E$, $E!\alpha$, $E := \alpha$, $E\ \alpha \in C$*
5. *$\alpha \in C$ implies $\alpha!v$, $\alpha := v$, $\alpha\ v \in C$*
6. *$\alpha, \beta \in C$ implies $\alpha\|s\|\beta$, $\alpha\|s\|s'$, $s'\|s\|\alpha \in C$*
7. *$\alpha \in C$ implies $[\![\lambda id : \tau \bullet \alpha, \Gamma]\!]\ v \in C$*
8. *$\alpha \in C$ implies:*

   (a) let $x = \alpha$ in $E \in C$

(b) **if α then E_1 else $E_2 \in \mathcal{C}$**

where E, E_1, E_2 denote expressions, s, s' denote stores and v denote a computable value.

We will use $\alpha, \alpha', ..., \beta, \beta', ...$ to denote configurations drawn from $\mathcal{C}$.

The operational semantics is presented in the usual way, by defining a labeled transition system on configurations. There are two kinds of events, ranged over respectively by a and ϵ:

- Visible events: They consist in input events of the form $\langle ?, c, v, s\rangle$ and output events of the form $\langle !, c, v, s\rangle$ where c is a channel and v is a value in $\mathcal{V}$ and s is the current operational dynamic store. We will use the notation $\bar{a}$ to denote the complement action of a. For instance, the complement of $\langle ?, c, v, s\rangle$ is $\langle !, c, v, s\rangle$. Notice also that $\bar{\bar{a}}$ is a.
- A silent event noted ϵ that is used to denote internal moves such as synchronizations on complementary actions.

We will use $\diamond, \diamond', ...$ as events drawn from the set of visible and invisible events. We will write $\alpha \overset{\diamond}{\rightarrow} \beta$ to denote the evolution of α into β after performing the event $\diamond$.

The transition relation is defined as the smallest relation satisfying the axioms and rules given in the figures 2 and 3.

6 Consistency Theorem

In this section the intention is to prove that the static semantics is consistent w.r.t. the dynamic semantics. The primary objective underlying the consistency theorem is to ensure that an expression and the value it evaluates into, have the same type. Its ensures also that the evaluation of an expression only leads to observable effects of the store that are compatible with that of its original static effect. But we have also to handle some additional problems:

- An expression does not, due the presence of the recursion operator, necessarily terminate, and does thus not necessarily evolve into a value. We want the consistency theorem to establish consistency in these cases too.
- We also want to treat communication effects. We thus want to ensure that the evaluation of an expression only leads to observable communication effects that are compatible with that of its original static effect.
- We finally have, due to communication, to handle open systems that will potentially receive values from the outside, and send values to the outside. We thus have to consider only correctly typed inputing values, and verify that outputing values are conform to the channel types.

Theorem 6.1 (Consistency)
Let α be a configuration, suppose that $\mathcal{S}, \mathcal{K} \models store(\alpha) : \sigma, \kappa$ and $store(\alpha) : \sigma, \kappa, \mathcal{S}, \mathcal{K} \models \Gamma : \mathcal{E}$. If $\mathcal{E} \vdash expr(\alpha) : \tau, \sigma', \kappa'$ and $\Gamma \vdash \alpha \overset{\diamond}{\rightarrow} \alpha'$ then, provided that whenever $\diamond$ is an input event its value is conform to the type of the involved channel (i.e. whenever $\diamond = \langle ?, c, v, s\rangle$ for some channel c and some value v, then $s : \sigma, \kappa, \mathcal{S}, \mathcal{K} \models v : \tau_1$ and $s : \sigma, \kappa, \mathcal{S}, \mathcal{K} \models c : chan_{\mathcal{K}(c)}(\tau_1)$), there exist $\mathcal{S}'$ and $\mathcal{K}'$ extending $\mathcal{S}$ and $\mathcal{K}$, and unobservable effects σ'' and κ'' i.e. $Observe(\mathcal{E}, \tau, \sigma'') = \emptyset$ and $Observe(\mathcal{E}, \tau, \kappa'') = \emptyset$, such that:

- *If $exp(\alpha')$ is a value then:*
  1. *$\mathcal{S}', \mathcal{K}' \models store(\alpha') : \sigma \cup \sigma' \cup \sigma'', \kappa \cup \kappa' \cup \kappa''$ and,*
  2. *$store(\alpha') : \sigma \cup \sigma' \cup \sigma'', \kappa \cup \kappa' \cup \kappa'', \mathcal{S}', \mathcal{K}' \models expr(\alpha') : \tau$ and,*
  3. *$\mathcal{K}' \models \diamond : \kappa' \cup \kappa''$*
- *Else there exist $\sigma_1', \sigma_2', \kappa_1'$ and κ_2', such that:*

  $\sigma_1' \cup \sigma_2' = \sigma'$ and
  $\kappa_1' \cup \kappa_2' = \kappa'$ and
  $\mathcal{S}', \mathcal{K}' \models store(\alpha') : \sigma \cup \sigma_1' \cup \sigma'', \kappa \cup \kappa_1' \cup \kappa''$
  and $\mathcal{K}' \models \diamond : \kappa_1' \cup \kappa''$ and
  $\mathcal{E} \vdash expr(\alpha') : \tau, \sigma_2', \kappa_2'$ and
  $store(\alpha') : \sigma \cup \sigma_1' \cup \sigma''$,
  $\kappa \cup \kappa_1' \cup \kappa'', \mathcal{S}', \mathcal{K}' \models \Gamma : \mathcal{E}$

Furthermore if $\diamond$ is an output event, its value is conform to the type of the involved channel.

7 Conclusion

We have reported in this paper the complete definition of an implicitly strongly typed polymorphic concurrent and functional language that supports data accepting in-place modification. We have presented a complete static semantics that rests on an extension of the type and effect discipline to handle communication effects. Afterwards we have presented an operational semantics of the language that rests on an extension of VPLA operational semantics. The consistency of the typing system w.r.t. the operational semantics have been established.

As a future research, we plan to investigate refinement issues as well as structuring and modularity mechanisms. We are particularly interested in experimenting some new approaches in modularity from the algebraic specification world such as the loose stratified semantics proposed by [Bid89]. Another important research interest for us is to develop an axiomatic semantics of our language as well as its mechanisation in order to prove program properties.

REFERENCES

[BB91] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the seventeenth ACM Symposium* *on Principles of Programming Languages*, 1991.

[BD92] D. Bolignano and M. Debabi. Higher order communicating processes with value-passing, assignment and return of results. In *Proceedings of the ISAAC'92 Conference, LNCS 650*. Springer Verlag, December 1992.

[BD93] D. Bolignano and M. Debabi. A denotational model for the integration of concurrent functional and imperative programming. In *Proceedings of the ICCI'93*. IEEE, May 1993.

[Ber89] B. Berthomieu. Implementing CCS, the LCS experiment. Technical Report 89425, LAAS CNRS, 1989.

[BHR84] S.D. Brooks, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *ACM*, 31(3):560–599, July 1984.

[Bid89] M. Bidoit. *PLUSS, un Langage pour le Développement de Spécifications Algébriques Modulaires*. PhD thesis, Paris Sud, July 1989.

[BMT92] D. Berry, A.J.R.G. Milner, and D. Turner. A semantics for ML concurrency primitives. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1992.

[GJLS87] D.K. Gifford, P. Jouvelot, J.M. Lucassen, and M.A. Sheldon. Fx-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[Hen85] M. Hennessy. Acceptance trees. *ACM*, 32:896–928, October 1985.

[HI90] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value passing. In *Proc. 17th International Colloquium on Automata, Languages and Programming, LNCS*. Springer Verlag, 1990.

[HI91] M. Hennessy and A. Ingólfsdóttir. Communicating processes with value-passing and assignments. Technical report, University of Sussex - Draft, June 1991.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1988.

[LT92] L. Leth and B. Thomsen. Some Facile chemistry. Technical Report ECRC-92-14, European Computer-Industry Research Center, 1992.

[Luc87] J.M. Lucassen. *Type and Effects: Towards an Integration of Functional and Imperative Programming*. PhD thesis, Laboratory of Computer Science, MIT, 1987.

[LW91] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proceedings of the seventeenth ACM Symposium on Principles of Programming Languages*, 1991.

[Mil78] A.J.R.G. Milner. A theory of type polymorphism in programming. *Computer and systems sciences*, 17:348–375, 1978.

[Mil89] A.J.R.G. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MTH90] A.J.R.G. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Rep91] J.H Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Department of Computer Science, Cornell University, August 1991.

[ST85] D. Sannella and A. Tarlecki. Program specification and development in standard ML. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, 1985.

[TJ92] J. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. Logic in Computer Science*, 1992.

[Tof87] M. Tofte. *Operational semantics and polymorphic type inference*. PhD thesis, Department of Computer Science, University of Edinburgh, 1987.

8 Appendix

(unit) $\mathcal{E} \vdash () : Unit, \emptyset, \emptyset$

(true) $\mathcal{E} \vdash \mathbf{true} : Bool, \emptyset, \emptyset$

(false) $\mathcal{E} \vdash \mathbf{false} : Bool, \emptyset, \emptyset$

(num) $\mathcal{E} \vdash Number\ n : Int, \emptyset, \emptyset$

(skip) $\mathcal{E} \vdash \mathbf{skip} : Unit, \emptyset, \emptyset$

(var) $$\frac{\tau \prec \mathcal{E}(x)}{\mathcal{E} \vdash x : \tau, \emptyset, \emptyset}$$

(abs) $$\frac{\mathcal{E}_x \dagger [x \mapsto \tau'] \vdash E : \tau, \sigma, \kappa}{\mathcal{E} \vdash \lambda x \bullet E : \tau' \xrightarrow{\sigma,\kappa} \tau, \emptyset, \emptyset}$$

(app) $$\frac{\mathcal{E} \vdash E : \tau \xrightarrow{\sigma,\kappa} \tau', \sigma', \kappa' \quad \mathcal{E} \vdash E' : \tau, \sigma'', \kappa''}{\mathcal{E} \vdash (E\ E') : \tau', \sigma \cup \sigma' \cup \sigma'', \kappa \cup \kappa' \cup \kappa''}$$

(op) $$\frac{\mathcal{E} \vdash E : \tau, \sigma, \kappa \quad \mathcal{E} \vdash E' : \tau, \sigma', \kappa'}{\mathcal{E} \vdash E\ op\ E' : \tau, \sigma \cup \sigma', \kappa \cup \kappa'}$$

where: op= $\sqcap, [\,], \|$

(ref) $$\frac{\mathcal{E} \vdash E : \tau, \sigma, \kappa}{\mathcal{E} \vdash \mathbf{ref}\ E : ref_\rho(\tau), \sigma \cup \mathbf{init}(\rho, \tau), \kappa}$$

(deref) $$\frac{\mathcal{E} \vdash E : ref_\rho(\tau), \sigma, \kappa}{\mathcal{E} \vdash !E : \tau, \sigma \cup read(\rho), \kappa}$$

(ass) $$\frac{\mathcal{E} \vdash E : ref_\rho(\tau), \sigma, \kappa \quad \mathcal{E} \vdash E' : \tau, \sigma', \kappa'}{\mathcal{E} \vdash E := E' : Unit, \sigma \cup \sigma' \cup write(\rho), \kappa \cup \kappa'}$$

(chan) $\mathcal{E} \vdash \mathbf{channel}() : chan_\chi(\tau), \emptyset, chan(\chi, \tau)$

(in) $$\frac{\mathcal{E} \vdash E : chan_\chi(\tau), \sigma, \kappa}{\mathcal{E} \vdash E? : \tau, \sigma, \kappa \cup in(\chi)}$$

(out) $$\frac{\mathcal{E} \vdash E : chan_\chi(\tau), \sigma, \kappa \quad \mathcal{E} \vdash E' : \tau, \sigma', \kappa'}{\mathcal{E} \vdash E!E' : Unit, \sigma \cup \sigma', \kappa \cup \kappa' \cup out(\chi)}$$

(seq) $$\frac{\mathcal{E} \vdash E : \tau, \sigma, \kappa \quad \mathcal{E} \vdash E' : \tau', \sigma', \kappa'}{\mathcal{E} \vdash E\ ;\ E' : \tau', \sigma \cup \sigma', \kappa \cup \kappa'}$$

(if)
$$\frac{\mathcal{E} \vdash E : Bool, \sigma, \kappa \quad \mathcal{E} \vdash E' : \tau, \sigma', \kappa' \quad \mathcal{E} \vdash E'' : \tau, \sigma'', \kappa''}{\mathcal{E} \vdash \mathbf{if}\ E\ \mathbf{then}\ E'\ \mathbf{else}\ E'' : \tau, \sigma \cup \sigma' \cup \sigma'', \kappa \cup \kappa' \cup \kappa''}$$

(let)
$$\frac{\mathcal{E} \vdash E' : \tau', \sigma', \kappa' \quad \mathcal{E}_x \dagger [x \mapsto Gen(\sigma', \kappa', \mathcal{E})(\tau')] \vdash E : \tau, \sigma, \kappa}{\mathcal{E} \vdash \mathbf{let}\ x = E'\ \mathbf{in}\ E : \tau, \sigma \cup \sigma', \kappa \cup \kappa'}$$

(rec) $$\frac{\mathcal{E}_x \dagger [x \mapsto \tau] \vdash E : \tau, \sigma, \kappa}{\mathcal{E} \vdash \mathbf{rec}\ x \bullet E : \tau, \sigma, \kappa}$$

(obs)
$$\frac{\mathcal{E} \vdash E : \tau, \sigma, \kappa \quad \sigma' \sqsupseteq Observe(\mathcal{E}, \tau, \sigma) \quad \kappa' \sqsupseteq Observe(\mathcal{E}, \tau, \kappa)}{\mathcal{E} \vdash E : \tau, \sigma', \kappa'}$$

Table 1: The static semantics

Constants:

$$\frac{\Box}{\Gamma \vdash (\mathbf{skip}, s) \xrightarrow{\epsilon} ((), s)}$$

Configuration Fork:

$$\frac{\Box}{\Gamma \vdash (E_1 \ op \ E_2, s) \xrightarrow{\epsilon} (E_1, s) op (E_2, s)} \quad op = \sqcap, \Box$$

Id Evaluation:

$$\frac{\Box}{\Gamma \dagger [x \mapsto v] \vdash (x, s) \xrightarrow{\epsilon} (v, s)}$$

Internal Choice:

$$\frac{\Box}{\begin{array}{lcl} \Gamma \vdash \alpha \sqcap \beta & \xrightarrow{\epsilon} & \alpha \\ & \xrightarrow{\epsilon} & \beta \end{array}}$$

External Choice:

$$\frac{\Gamma \vdash \alpha \xrightarrow{a} \alpha'}{\begin{array}{lcl} \Gamma \vdash \alpha \Box \beta & \xrightarrow{a} & \alpha' \\ \beta \Box \alpha & \xrightarrow{a} & \alpha' \end{array}}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\epsilon} \alpha'}{\begin{array}{lcl} \Gamma \vdash \alpha \Box \beta & \xrightarrow{\epsilon} & \alpha' \Box \beta \\ \beta \Box \alpha & \xrightarrow{\epsilon} & \beta \Box \alpha' \end{array}}$$

$$\frac{\Box}{\begin{array}{lcl} \Gamma \vdash (v, s) \Box \alpha & \xrightarrow{\epsilon} & (v, s) \\ \alpha \Box (v, s) & \xrightarrow{\epsilon} & (v, s) \end{array}}$$

Parallel Composition:

$$\frac{\Box}{\Gamma \vdash (E_1 \| E_2, s) \xrightarrow{\epsilon} (E_1, s) \| s \| (E_2, s)}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{a} \alpha', \ \Gamma \vdash \beta \xrightarrow{\bar{a}} \beta'}{\begin{array}{lcl} \Gamma \vdash \alpha \| s \| \beta & \xrightarrow{\epsilon} & \alpha' \| s \| \beta' \\ \beta \| s \| \alpha & \xrightarrow{\epsilon} & \beta' \| s \| \alpha' \end{array}}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\begin{array}{lcl} \Gamma \vdash \alpha \| s \| \beta & \xrightarrow{\diamond} & \alpha' \| s \| \beta \\ \beta \| s \| \alpha & \xrightarrow{\diamond} & \beta \| s \| \alpha' \end{array}}$$

$$\frac{\Box}{\begin{array}{lcl} \Gamma \vdash \alpha \| s \| (v, s') & \xrightarrow{\epsilon} & \alpha \| s \| s' \\ (v, s') \| s \| \alpha & \xrightarrow{\epsilon} & s' \| s \| \alpha \end{array}}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\begin{array}{lcl} \Gamma \vdash \alpha \| s \| s' & \xrightarrow{\diamond} & \alpha' \| s \| s' \\ s' \| s \| \alpha & \xrightarrow{\diamond} & s' \| s \| \alpha' \end{array}}$$

$$\frac{\Box}{\begin{array}{lcl} \Gamma \vdash (v, s'') \| s \| s' & \xrightarrow{\epsilon} & (v, merge(s, s', s'')) \\ s' \| s \| (v, s'') & \xrightarrow{\epsilon} & (v, merge(s, s', s'')) \end{array}}$$

Sequencing:

$$\frac{\Box}{\Gamma \vdash (E_1 ; E_2, s) \xrightarrow{\epsilon} (E_1, s) ; E_2}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \alpha ; E \xrightarrow{\diamond} \alpha' ; E}$$

$$\frac{\Box}{(v, s) ; E \xrightarrow{\epsilon} (E, s)}$$

$$\frac{\Box}{\Gamma \vdash ((v, s), E) \xrightarrow{\epsilon} (v, (E, s))}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash (v, \alpha) \xrightarrow{\diamond} (v, \alpha')}$$

Function:

$$\frac{\Box}{\Gamma \vdash (\lambda x \bullet E, s) \xrightarrow{\epsilon} (\llbracket \lambda x \bullet E, \Gamma \ \rrbracket, s)}$$

$$\frac{\Box}{\Gamma \vdash (E_1 \ E_2, s) \xrightarrow{\epsilon} E_1 \ (E_2, s)}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash E \ \alpha \xrightarrow{\diamond} E \ \alpha'}$$

$$\frac{\Box}{\Gamma \vdash E \ (v, s) \xrightarrow{\epsilon} (E, s) \ v}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \alpha \ v \xrightarrow{\diamond} \alpha' \ v}$$

$$\frac{\Box}{\Gamma \vdash (\llbracket \lambda x \bullet E, \Gamma_1 \rrbracket, s) \ v \xrightarrow{\epsilon} \llbracket \lambda x \bullet (E, s), \Gamma_1 \rrbracket \ v}$$

$$\frac{\Gamma_1 \dagger [x \mapsto v] \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \llbracket \lambda x \bullet \alpha, \Gamma_1 \rrbracket \ v \xrightarrow{\diamond} \llbracket \lambda x \bullet \alpha', \Gamma_1 \rrbracket \ v}$$

$$\frac{\Gamma_1 \dagger [x \mapsto v] \vdash \alpha \xrightarrow{\diamond} (v', s)}{\Gamma \vdash \llbracket \lambda x \bullet \alpha, \Gamma_1 \rrbracket \ v \xrightarrow{\diamond} (v', s)}$$

Let Expressions:

$$\frac{\Box}{\begin{array}{c} \Gamma \vdash (\mathbf{let}\ x = E_1 \ \mathbf{in}\ E_2 \ , s) \quad \xrightarrow{\epsilon} \\ \mathbf{let}\ x = (E_1, s) \ \mathbf{in}\ E_2 \end{array}}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \mathbf{let}\ x = \alpha \ \mathbf{in}\ E \xrightarrow{\diamond} \mathbf{let}\ x = \alpha' \ \mathbf{in}\ E}$$

$$\frac{\Box}{\Gamma \vdash \mathbf{let}\ x = (v, s) \ \mathbf{in}\ E \xrightarrow{\epsilon} (E[v/x], s)}$$

Table 2: Operational semantics: Part I

If Expressions:

$$\frac{\Box}{\Gamma \vdash (\mathbf{if}\ E\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2\ , s) \xrightarrow{\epsilon} \mathbf{if}\ (E, s)\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \mathbf{if}\ \alpha\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \xrightarrow{\diamond} \mathbf{if}\ \alpha'\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2}$$

$$\frac{\Box}{\Gamma \vdash \mathbf{if}\ (\mathbf{true}, s)\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \xrightarrow{\epsilon} (E_1, s)}$$

$$\frac{\Box}{\Gamma \vdash \mathbf{if}\ (\mathbf{false}, s)\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \xrightarrow{\epsilon} (E_2, s)}$$

Recursion:

$$\frac{\Box}{\Gamma \vdash (\mathbf{rec}\ x \bullet E, s) \xrightarrow{\epsilon} (E[\mathbf{rec}\ x \bullet E/x], s)}$$

Channel Creation:

$$\frac{\Box}{\Gamma \vdash (\mathbf{channel}(), s) \xrightarrow{\epsilon} (c, s \cup \{chan(c)\})} \quad chan(c) \notin s$$

Reference Creation:

$$\frac{\Box}{\Gamma \vdash (\mathbf{ref}(E), s) \xrightarrow{\epsilon} \mathbf{ref}\ (E, s)}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \mathbf{ref}\ \alpha \xrightarrow{\diamond} \mathbf{ref}\ \alpha'}$$

$$\frac{\Box}{\Gamma \vdash \mathbf{ref}\ (v, s) \xrightarrow{\epsilon} (r, s \cup \{init(r, v)\})} \quad \nexists v \bullet init(r, v) \in s$$

Input:

$$\frac{\Box}{\Gamma \vdash (E?, s) \xrightarrow{\epsilon} (E, s)?}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \alpha? \xrightarrow{\diamond} \alpha'?}$$

$$\frac{\Box}{\Gamma \vdash (c, s)? \xrightarrow{(?, c, v, s')} (v, pass(s', s, v))}$$

Output:

$$\frac{\Box}{\Gamma \vdash (E_1 ! E_2, s) \xrightarrow{\epsilon} E_1 !(E_2, s)}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash E!\alpha \xrightarrow{\diamond} E!\alpha'}$$

$$\frac{\Box}{\Gamma \vdash E!(v, s) \xrightarrow{\epsilon} (E, s)!v}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \alpha!v \xrightarrow{\diamond} \alpha'!v}$$

$$\frac{\Box}{\Gamma \vdash (c, s)!v \xrightarrow{(!, c, v, s)} ((), s)}$$

Dereferencing:

$$\frac{\Box}{\Gamma \vdash (!E, s) \xrightarrow{\epsilon} !(E, s)}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash !\alpha \xrightarrow{\diamond} !\alpha'}$$

$$\frac{\Box}{\Gamma \vdash !(r, s) \xrightarrow{\epsilon} (v, s \cup \{read(r)\})} \quad init(r, v) \in s$$

Assignment:

$$\frac{\Box}{\Gamma \vdash (E_1 := E_2, s) \xrightarrow{\epsilon} E_1 := (E_2, s)}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash E := \alpha \xrightarrow{\diamond} E := \alpha'}$$

$$\frac{\Box}{\Gamma \vdash E := (v, s) \xrightarrow{\epsilon} (E, s) := v}$$

$$\frac{\Gamma \vdash \alpha \xrightarrow{\diamond} \alpha'}{\Gamma \vdash \alpha := v \xrightarrow{\diamond} \alpha' := v}$$

$$\frac{\Box}{\Gamma \vdash (r, s) := v \xrightarrow{\epsilon} ((), s_r \cup \{write(r),\ init(r, v)\})}$$

Table 3: Operational Semantics

Peirce Algebras*

Chris Brink[†] Katarina Britz[†]
Renate A. Schmidt[†§]

[†] *Department of Mathematics, University of Cape Town,
Rondebosch 7700, South Africa*
[§] *Max-Planck-Institut für Informatik, Im Stadtwald,
W-6600 Saarbrücken 11, Germany*

In its modern form the algebra of relations has been under investigation by mathematicians since Tarski's seminal (1941) paper. The main line of development has been the study of a class of algebras called *relation algebras* (Chin and Tarski 1951, Jónsson 1982), in parallel with developments such as Boolean algebras with operators (Jónsson and Tarski 1951/1952) and cylindric algebras (Henkin, Monk and Tarski 1985). Since the early seventies the algebra of relations has increasingly become of interest to computer scientists. Just as the notion of a partial function provides a natural model for deterministic programs, so the more general notion of a (binary) relation provides a natural model for nondeterministic programs. This idea has been exploited by various authors. For example, it is evident in Floyd-Hoare logic for program verification, it has been extended to specification in Hoare and He, Jifeng (1987), it figures in logics of programs such as dynamic logic (Parikh 1981, Harel 1984), and it was used in the early seventies to model recursive procedures (de Bakker and de Roever 1973, Hitchcock and Park 1972). Recently the algebra of relations has been extensively used in a graph-theoretic approach to programs by Schmidt and Ströhlein (1991). In modal logic, relation algebra features strongly in the Dutch-Hungarian cooperation on van Benthem's (1991) new *arrow logic* (see *Logic at Work, Proceedings of the Applied Logic Conference* (1992)). Venema (1992) is another interdisciplinary study of relation algebra and multi-modal logic. The proof theory of relations is also of interest to computer scientists, and several relational inference systems are available (Wadge 1975, Hennessy 1980, Maddux 1983, Orlowska 1991).

In many applications it has become clear that we need, not just an algebra of relations as distinct from an algebra of sets, but an algebra of relations *interacting* with sets. (For example, if we view a program as effecting a transition on a state space, we may wish to model this by a binary relation acting on a set of states.) Such an algebra was presented in Brink (1981) under the name of *Boolean modules*. A Boolean module is defined (Brink 1988) as a two-sorted algebra $\mathcal{M} = (\mathcal{B}, \mathcal{R}, :)$, where $\mathcal{B}$ is a Boolean algebra, $\mathcal{R}$ is a relation algebra and $:$ is a mapping $\mathcal{R} \times \mathcal{B} \longrightarrow \mathcal{B}$ written $r:a$ such that for any $r, s \in \mathcal{R}$ and $a, b \in \mathcal{B}$:

M1 $r:(a+b) = r:a + r:b$
M2 $(r+s):a = r:a + s:a$
M3 $r:(s:a) = (r\,;s):a$
M4 $e:a = a$
M5 $0:a = 0$
M6 $r^{\smile}:(r:a)' \leq a'$.

*To appear in *Formal Aspects of Computing.*

The symbols $+$, $:$, $;$, ϵ, 0, $^{-}$, $'$ and $\leq$ respectively denote join, Peirce product, relational composition, identity, zero, converse, complementation and the usual partial ordering. Let A be any subset of some non-empty set U and let R, S be any binary relations over U. In the standard models (i.e., in *proper Boolean modules*) the join is set-theoretic union. The *Peirce product* $R:A$ is the set of elements x related by R to some element y in A. The relational composition $R;S$ is the set of pairs (x,y) for which there is a z such that $(x,z) \in R$ and $(z,y) \in S$. The identity is the identity relation over U. The zero is the empty set. The converse of a relation R is the set R^{-} of pairs (y,x) for which $(x,y) \in R$. Complementation of sets (respectively relations) is with respect to U (respectively $U \times U$). And, $\leq$ is interpreted as the subset relation.

Though independent of the computer science context, Boolean modules are very similar to *dynamic algebras*, introduced by Kozen (1980) as the algebraic version of dynamic logic. And both of these are quite similar to the *extended relation algebras* introduced by Suppes (1976) in a linguistic context. However, Boolean modules and dynamic algebras both have the drawback of not treating relations (programs) and sets equally: there is a set-forming operator on relations, but no relation-forming operator on sets. Extended relation algebras do not have this drawback, but they do have the drawback of being as yet unformalized as algebras.

We present here a two-sorted algebra, called a *Peirce algebra*, of relations and sets interacting with each other. In a Peirce algebra, sets (or rather, the variables representing sets) can combine with each other as in a Boolean algebra, relations can combine with each other as in a relation algebra, and in addition we have both a set-forming operator on relations and a relation-forming operator on sets. The former is the Peirce product used in Boolean modules; the latter is the operation of *cylindrification*. Peirce algebras thus present a natural next step after Boolean algebras, relation algebras and Boolean modules.

Formally, we define a *Peirce algebra* to be a Boolean module $(\mathcal{B}, \mathcal{R}, :)$ enriched with an operation ^{c} from the underlying Boolean algebra $\mathcal{B}$ to the underlying relation algebra $\mathcal{R}$ such that for every $a \in \mathcal{B}$ and $r \in \mathcal{R}$:

P1 $a^{c}:1 = a$

P2 $(r:1)^{c} = r;1$.

In the standard models (i.e., in *proper Peirce algebras*) applying the cylindrification operation to a set A yields the relation A^{c} given by the Cartesian product $A \times U$. An example of a Peirce algebra is any extended relation algebra. Another example is any relation algebra. We show that the underlying Boolean algebra $\mathcal{B}$ of any Peirce algebra can be embedded in its underlying relation algebra $\mathcal{R}$ in two ways: as the Boolean algebra of so-called right ideal elements in $\mathcal{R}$, and as the Boolean algebra of elements below the identity of $\mathcal{R}$. These results reiterate the point made by Maddux (1990) that Peirce algebra is not a mathematical requisite for modelling interactions between relations and sets, in the sense that these can be modelled in relation algebras (as interactions with right ideal elements, for example). However, we argue that Peirce algebra provides a more natural framework for doing so. In a Peirce algebra one can actually manipulate both sets and relations simultaneously. From an applications-oriented point of view this is an advantage, and we present two (sets of) sample applications to substantiate this point.

The first shows how three programming constructs in the calculus of weakest prespecification of Hoare and He, Jifeng (1987) can be modelled naturally in Peirce algebras. This comes about through the isomorphism in any Peirce algebra $(\mathcal{B}, \mathcal{R}, :, ^{c})$ between the

Boolean algebra $\mathcal{B}$ and the Boolean algebra of right ideal elements of the relation algebra $\mathcal{R}$ and the isomorphism between $\mathcal{B}$ and the Boolean algebra of identity elements in $\mathcal{R}$. First, Hoare and He, Jifeng (1987) use right ideal elements to model conditional statements in logics representing programs as binary relations. Second, subsets of the identity relation are used to model a test operation (Parikh 1981). Third, left ideal elements can be used to model the initialization of abstract data types as defined in Hoare, He, Jifeng and Sanders (1987).

The second application points out that the so-called *terminological logics* arising in knowledge representation based on the system KL-ONE (Woods and Schmolze 1992) have evolved a semantics best described as a calculus of relations interacting with sets. Brink and Schmidt (1992) show that the terminological representation language $\mathcal{ALC}$ of Schmidt-Schauß and Smolka (1991) can be captured in the context of Boolean modules. In this paper we extend this idea and use Peirce algebra to accommodate terminological representation languages even more expressive than $\mathcal{ALC}$.

Terminological representation languages have two syntactic primitives, called *concepts* and *roles*. Concepts are usually interpreted as sets and roles as binary relations. As sets and relations have simple calculi that can be presented, respectively, in the context of Boolean algebra and relation algebra, concepts can be modelled in Boolean algebra and roles in relation algebra. Concepts and roles also interact in certain ways, and these can be modelled as interactions between relations and sets. More specifically, concept-forming operations on roles can be interpreted as variants of Peirce product (with two exceptions), and an algebraic characterization for such interactions are Boolean modules. (The exceptions involve numerical quantification.) Role-forming operators on concepts can be interpreted in terms of cylindrification. A natural algebraic presentation for such interactions is then Peirce algebra. The advantages for doing so are: First, Peirce algebra provides a formal mathematical framework for KL-ONE-based knowledge representation, the development of which has, by and large, been implementation-driven and rather *ad hoc*. Second, Peirce algebra provides a natural (equational) axiomatization for reasoning about information represented in a terminological language. Third, terminological representation can be linked to other areas of application of Peirce algebra. Schmidt (1993), for example, exploits the link between Peirce algebra and extended relation algebra and shows how terminological representation can benefit from Suppes' (1976) linguistic analysis of English language sentences.

References

Brink, C. (1981), Boolean modules, *Journal of Algebra* **71**(2), 291–313.

Brink, C. (1988), On the application of relations, *S. Afr. J. Philos.* **7**(2), 105–112.

Brink, C. and Schmidt, R. A. (1992), Subsumption computed algebraically, *Computers and Mathematics with Applications* **23**(2–9), 329–342.

Chin, L. H. and Tarski, A. (1951), Distributive and modular laws in the arithmetic of relation algebras, *Univ. Calif. Publ. Math.* **1**(9), 341–384.

de Bakker, J. W. and de Roever, W. P. (1973), A calculus for recursive program schemes, *in* M. Nivat (ed.), *Symposium on Automata, Formal Languages and Programming*, North Holland, Amsterdam.

Harel, D. (1984), Dynamic logic, *in* D. Gabbay and F. Guenther (eds), *Handbook of Philosophical Logic*, Vol. II, Reidel Publ. Co., Dordrecht, Holland, pp. 497–604.

Henkin, L., Monk, J. D. and Tarski, A. (1985), *Cylindric Algebras: Part II*, Vol 115 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, Amsterdam.

Hennessy, M. C. B. (1980), A proof-system for the first-order relational calculus, *Journal of Computer and System Sciences* **20**, 96–110.

Hitchcock, P. and Park, D. (1972), Induction rules and termination proofs, *in* M. Nivat (ed.), *Automata, Languages and Programming*, North-Holland, Amsterdam.

Hoare, C. A. R. and He, Jifeng (1987), The weakest prespecification, *Information Processing Letters* **24**, 127–132.

Hoare, C. A. R., He, Jifeng and Sanders, J. W. (1987), Prespecification in data refinement, *Information Processing Letters* **25**, 71–76.

Jónsson, B. (1982), Varieties of relation algebras, *Algebra Universalis* **15**(3), 273–298.

Jónsson, B. and Tarski, A. (1951/1952), Boolean algebras with operators, Part I/II, *American Journal of Mathematics* **73/74**, 891–939/127–162.

Kozen D. (1980), A representation theorem for models of ∗-free PDL, *in* J. De Bakker and J. van Leeuwen (eds), *Automata, Languages and Programming*, Vol. 85 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 351–362.

Logic at Work, Proceedings of the Applied Logic Conference (1992), University of Amsterdam, Amsterdam. Preprint. To appear.

Maddux, R. D. (1983), A sequent calculus for relation algebras, *Annals of Pure and Applied Logic* **25**, 73–101.

Maddux, R. D. (1990). Personal communication with C. Brink.

Orlowska, E. (1991), Relational interpretation of modal logic, *in* H. Andréka, J. D. Monk and I. Németi (eds), *Algebraic Logic*, Vol. 54 of *Colloquia Mathematica Societatis János Bolyai*, North-Holland, Amsterdam, pp. 443–471.

Parikh, D. (1981), Propositional dynamic logic of programs: A survey, *in* E. Engeler (ed.), *Logic of Programs*, Vol. 125 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 102–144.

Schmidt, G. and Ströhlein, T. (1991), *Relations and Graphs*, Springer-Verlag, Berlin.

Schmidt, R. A. (1993), Terminological representation, natural language & relation algebra, *in* H. J. Ohlbach (ed.), *Proceedings of the sixteenth German AI Conference (GWAI-92)*, Vol. 671 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, pp. 357–371.

Schmidt-Schauß, M. and Smolka, G. (1991), Attributive concept description with complements, *Artificial Intelligence* **48**, 1–26.

Suppes, P. (1976), Elimination of quantifiers in the semantics of natural language by use of extended relation algebras, *Rev. Int. de Philosophie* **30**(3–4), 243–259.

Tarski, A. (1941), On the calculus of relations, *Journal of Symbolic Logic* **6**(3), 73–89.

van Benthem, J. (1991), Logic and the flow of information, *Technical Report, ILLC Prepublication Series for Logic, Semantics and Philosophy of Language LP-92-11*, Institute for Logic, Language and Computation, University of Amsterdam, Amsterdam. To appear.

Venema, Y. (1992), *Many-Dimensional Modal Logic*, PhD thesis, University of Amsterdam, Amsterdam.

Wadge, W. W. (1975), A complete natural deduction system for the relational calculus, *Theory of Computation Report 5*, University of Warwick.

Woods, W. A. and Schmolze, J. G. (1992), The KL-ONE family, *Computers and Mathematics with Applications* **23**(2–5), 133–177.

Comparing Two Different Approaches to Products in Abstract Relation Algebras

R. Berghammer[*] A. Haeberer[†] G. Schmidt[*] P. Veloso[†]

1 Introduction

The study of relation algebras has its roots in the second half of the XIX century with the pioneering work of Boole and de Morgan. Later on, Peirce in a series of papers developed the algebra of relations, and by the end of the century Schröder definitively set the basis of modern relation algebra in his magnum opus. The modern development of the topic starts with the fundamental work of Tarski and his co-workers (see e.g., [13, 5, 8]). In the early 70's relations and relational calculi began to be used for formal programming by de Bakker and de Roever. In the following decade, Hoare and He related the work of Birkhoff on residuals with Dijkstra's weakest precondition approach to programming. Recently, Möller used n-ary higher-order relations between nested tuples as elements of a language in which to specify and develop programs and Backhouse et al. developed a theory of data types based on the calculus of relations.

During the development of relation algebras as a formal programming tool, the need of some form of "categorical product" of relations became apparent, whether as a type or as an operation. This need was motivated by the lack of variables over individuals, which by itself is one of the main advantages of functional and abstract relational approaches to program development. Two approaches to this kind of extension arouse in the late 70's and the early 80's, which will be referred to as the "Munich approach" [10, 3] and the "Rio approach" [7, 15]. Both of them rely on relation algebras as presented by Chin and Tarski [5]. The former uses heterogeneous relations and undertakes the "product-extension" as being a data type by axiomatically introducing two projections π and ρ and defining the product in terms of them. The latter uses homogeneous relations and introduces axioms for a *fork* operation ∇, thus extending relation algebra in the same way Jónsson and Tarski in [8] extended a Boolean algebra by means of operators in order to obtain a relation algebra. The introduction of ∇ induces a free groupoid structure in the basic set of the relation standard model of relation algebra, which by allowing the internalization of relations poses some interesting representability questions [1].

The Munich group started from giving relational semantics to programming language constructs and constructing semantic domains by relation algebraic means. They worked with heterogeneous relation algebras [12], introduced the point axiom [11] for these and discussed how representability depends on it. Defining the symmetric quotient [2, 16] made it possible to handle set and function comprehension.

The Rio approach, motivated mainly by the development of a relational programming calculus not bounded by lack of expressiveness, first tackled the problem — posed in [13] and formally treated in [9] — of the impossibility of expressing first-order formulae with four or more variables in abstract relation algebra. As it was shown in [14], the expressive power of the Boolean algebra with operators resulting from the extension of relation algebra with the ∇-operation encompasses

[*] Fak. für Informatik, Universität der Bundeswehr, Werner-Heisenberg-Weg 39, D-85577 Neubiberg (Germany)

[†] Dep. de Informatica, Pontificia Univ. Católica, R. Marquês de S. Vicente 225, Rio de Janeiro, RJ 22453 (Brazil)

that of first-order logic. In using this calculus, they developed various case studies on formal program construction, see e.g., [15]. Some interesting work has been done concerning the problem of the smooth transformation by calculation of expressions universally quantified by means of the construction $\overline{RS}$ (the complement of the relational composition RS). Among other techniques under study, this was accomplished by the use of residuals in other ways than a straightforward solution X of inclusions of the form $XR \subset S$, which lead to a weakest precondition style of program construction.

This paper reports research work under joint development by the two groups. In this extended abstract we only compare the two relational approaches to products. The full paper also deals with some further topics like the formal construction of programs using abstract relation algebra.

2 The Munich Approach to Direct Products

Most operations occuring in real life involve several arguments and several sorts. Using relation algebra as a programming calculus, therefore, requires a means to deal with direct products and n-ary operations. In the following, the Munich approach to direct products is presented in a short manner. Also the monomorphy of the product is investigated.

In the Munich approach, direct products are characterized by means of natural projections, see [3, 2]. Then, one obtains the following specification (where I denotes the identity relation and L is the universal relation).

Definition. Given two relations π and ρ, we call the pair (π, ρ) a *direct product*, if

$$(1)\ \pi^{\mathrm{T}}\pi = \mathrm{I} \qquad (2)\ \rho^{\mathrm{T}}\rho = \mathrm{I} \qquad (3)\ \pi\pi^{\mathrm{T}} \cap \rho\rho^{\mathrm{T}} = \mathrm{I} \qquad (4)\ \pi^{\mathrm{T}}\rho = \mathrm{L}.$$

In this setting, π and ρ are called the *natural projections*. □

It is easy to verify that the projections from a Cartesian product $X \times Y$ to the components X and Y (considered as heterogeneous relations) are a model of (1) through (4). In this standard model we have: From (1) and (2) we get that the projections are univalent and surjective. Inclusion $\subset$ of the third axiom ensures that there is at most one pair with given images in X and Y; the other inclusion means that π and ρ are total, i.e., there are no "unprojected" pairs. And, finally, condition (4) describes the fact that for every $x \in X$ and $y \in Y$ the pair $\langle x, y \rangle$ is indeed contained in $X \times Y$.

We now investigate the question of how unique the direct product is determined by these rules. To this end, we need the following notions. Let R and S be two relations and consider a pair $\mathcal{H} := (\Phi, \Psi)$ of functions. $\mathcal{H}$ is called a *homomorphism* from R to S if $R \subset \Phi S \Psi^{\mathrm{T}}$ holds. If, in addition, the pair $(\Phi^{\mathrm{T}}, \Psi^{\mathrm{T}})$ is a homomorphism from S to R, then $\mathcal{H}$ is said to be an *isomorphism* between R and S. Therefore, an isomorphism (Φ, Ψ) between R and S is a pair of bijective functions Φ and Ψ, which satisfies the condition $R\Psi = \Phi S$.

By *purely relation-algebraic reasoning*, now, it can be shown that the direct product is uniquely characterized up to isomorphism:

Theorem 1. Assume that two direct products (π, ρ) and (π', ρ') are given together with two bijective functions Ψ_1 and Ψ_2 such that the products $\pi\Psi_1\pi'^{\mathrm{T}}$ and $\rho\Psi_2\rho'^{\mathrm{T}}$ are defined. Then the pair (Φ, Ψ_1), is an isomorphism between π and π' and the pair (Φ, Ψ_2), is an isomorphism between ρ and ρ', where the bijective function Φ is defined as $\Phi := \pi\Psi_1\pi'^{\mathrm{T}} \cap \rho\Psi_2\rho'^{\mathrm{T}}$. □

3 The Rio Approach to Direct Products

Now, we sketch the Rio approach to products. This approach is based on homogeneous relations and a fork-operation ∇ extending a relation algebra. The axioms of fork are as follows:

$$(5)\ R\nabla S = R(\mathrm{I}\nabla\mathrm{L}) \cap S(\mathrm{L}\nabla\mathrm{I}) \qquad (6)\ (R\nabla S)(P\nabla Q)^{\mathrm{T}} = RP^{\mathrm{T}} \cap SQ^{\mathrm{T}}.$$

It is a classical result that (homogeneous) relation algebra is inferior in expressive power to predicate logic. However, homogeneous relation algebra extended with a product using the operator ∇ and axioms (5) and (6) has the same expressive power as predicate logic [14].

4 Comparison

We are now going to compare the product axiomatization of the two approaches. In particular, we present a cross-derivation of either approach using the axiomatic presentation of the other. This is a little bit complicated as it means a comparison of results in a homogeneous and in a heterogeneous relation algebra. So we cannot in all cases expect textually identical results when simulating one feature in the other type of algebra.

First, we express π and ρ via ∇. To this end, let a ∇-extended homogeneous relation algebra be given. We consider partial identities, i.e., relations contained in the identity relation I. In the case of a relation on a set X each partial identity $\delta \subseteq X \times X$ describes a subset of X, viz. the set $\{x \in X : \langle x, x \rangle \in \delta\}$. It is easy to prove that partial identities are invariant under transposition and that composition of a partial identity with itself is idempotent. Thus, we are able to prove the following theorem defining the two projections in terms of the operation ∇.

Theorem 2. Let three partial identities δ_X, δ_Y, and δ be given satisfying the property

$$\delta = (\delta_X \nabla \mathsf{L})^{\mathrm{T}} (\delta_X \nabla \mathsf{L}) \cap (\mathsf{L} \nabla \delta_Y)^{\mathrm{T}} (\mathsf{L} \nabla \delta_Y).$$

Then it is possible to define two relations $\pi := (\delta_X \nabla \mathsf{L})^{\mathrm{T}}$ and $\rho := (\mathsf{L} \nabla \delta_Y)^{\mathrm{T}}$ such that the intentions of the above definition are met in the following form:

$$\pi^{\mathrm{T}} \pi = \delta_X \qquad \rho^{\mathrm{T}} \rho = \delta_Y \qquad \pi \pi^{\mathrm{T}} \cap \rho \rho^{\mathrm{T}} = \delta \qquad \pi^{\mathrm{T}} \rho = \delta_X \mathsf{L} \cap \mathsf{L} \delta_Y. \qquad \square$$

Now we deal with the other direction, i.e., the description of ∇ via the two projections. Therefore, we assume a heterogeneous relation algebra. Then we are able to prove:

Theorem 3. Let a direct product (π, ρ) be given. We define for this product an operation ∇ (in infix notation) by $R \nabla S := R \pi^{\mathrm{T}} \cap S \rho^{\mathrm{T}}$. Then we obtain the equation

$$R \nabla S = R(\mathrm{I} \nabla \mathsf{L}) \cap S(\mathsf{L} \nabla \mathrm{I}).$$

If the point axiom (see [11]) holds, i.e., the relation algebra is representable, then we also have

$$(R \nabla S)(P \nabla Q)^{\mathrm{T}} = R P^{\mathrm{T}} \cap S Q^{\mathrm{T}}. \qquad \square$$

One might ask whether the second statement of this theorem (of which "$\subset$" is rather trivial) could also proven without assuming the additional condition. Over the years we have tried very hard to solve this "unsharpness-problem". For example, in [3, 16] some weaker conditions than the point axiom can be found. Today, we believe that a proof of inclusion "$\supset$" without conditions is not possible, i.e., that there are models of the axioms of a heterogeneous relation algebra in which "$\neq$" holds.

We have shown that one approach may more or less directly simulate the other; so either one could be taken, the Rio approach with partial identities, as well as the Munich approach using heterogeneous relations. It should be mentioned that the major part of the proofs of the theorems were developed with RALF, a relation-algebraic formula manipulation system and proof checker developed in Munich [4] and re-implemented on a different hardware-software basis by the Rio group.

Finally, let us shortly discuss some advantages and disadvantages of the two approaches. While using partial identities, there are no problems with the applicability of operations. However, when nonfitting relations are multiplied, the result will often be the null relation. On the other hand, a supporting computer system such as RALF or RELVIEW should refuse to operate on nonfitting relations. Heterogeneous relations fit neatly into the way of thinking with sorts or types in Computer Science. Working with matrices and vectors might even let engineers feel comfortable since they are accustomed to them. A second difference between the two approaches is with respect to the existence of models. In the heterogeneous case, one can work with small models that certainly exist, such as the set of all boolean $n \times n, m \times m, n \times m, m \times n$-matrices. In contrast, already the very first examples in the other case is burdened with the question of whether the base set of all the partial identities is free of set-theoretical antinomies. There is also another important difference. When working with relations $R \subseteq X \times Y$ between sorts and types, one has the possibility of distinguishing the categorical object X from the domain $R\mathsf{L}$ where the relation is "defined". This difference, to which computer scientists are very much accustomed is usually hidden when using partial identities, since then one would have to manipulate *two* partial identities to fully handle R. In [6], pp. 334–354, however, a new kind of objects, called *problems* is studied taking this into account.

References

[1] G. Baum, A. Haeberer, P. Veloso: On the representability of the abstract relational algebra, IGPL Newsletter 1, 3 (September 1992) European Foundation for Logic, Language and Information Interest Group on Programming Logic

[2] R. Berghammer, G. Schmidt, H. Zierer: Symmetric quotients and domain constructions. Inform. Proc. Letters 33, 3, 163–168 (1989/90)

[3] R. Berghammer, H. Zierer: Relational algebraic semantics of deterministic and nondeterministic programs. TCS 43, 123–147 (1986)

[4] R. Brethauer: Ein Formelmanipulationssystem zur computergestützten Beweisführung in der Relationenalgebra, Universität der Bundeswehr München, Diplomarbeit (1991)

[5] L. Chin, A. Tarski: Distributive and modular laws in the arithmetic of relation algebras. University of California Publications in *Mathematics (new series) 1 (1951)*

[6] A. Haeberer, P. Veloso: Partial relations for program derivation: Adequacy, Inevitability and expressiveness. In: B. Möller (ed.), Constructing programs from specifications, North-Holland, 319–371 (1991)

[7] A. Haeberer, P. Veloso, P. Elustondo: Towards a relational calculus for software construction. Meeting of IFIP WG 2.1, Chester, England (1990)

[8] B. Jónsson, A. Tarski: Boolean algebras with operators, Part II. Amer. J. Math. 74, 127–167 (1952)

[9] R. Maddux: A sequent calculus for relation algebras. Ann. of Pure and Applied Logic 25, 73–101 (1983)

[10] G. Schmidt: Programs as partial graphs I: Flow equivalence and correctness. TCS 15, 1–25 (1981)

[11] G. Schmidt, T. Ströhlein: Relation algebras: Concept of points and representability. Discrete Math. 54, 83–92 (1985)

[12] G. Schmidt, T. Ströhlein: Relations and graphs. EATCS Monographs in CS, Springer (1993)

[13] A. Tarski: On the calculus of relations. J. Symbolic Logic 6, 73–89 (1941)

[14] P. Veloso, A. Haeberer: A finitary relational algebra for classical first-order logic. Bull. of the Section on Logic of the Polish Academy of Sciences 20, 52–62 (1991)

[15] P. Veloso, A. Haeberer, G. Baum: Formal program construction within an extended calculus of binary relations. J. Symbolic Comp. (to appear)

[16] H. Zierer: Relation algebraic domain constructions. TCS 87, 163–188 (1991)

Specifying Type Systems with Multi-Level Order-Sorted Algebra

Martin Erwig*

We show how to use order-sorted algebras on multiple levels to describe type systems and languages, in particular, data models and query languages. It is demonstrated that even advanced aspects can be modeled, including, parametric polymorphism, relationships between different sorts of an operation's rank, the specification of a variable number of parameters for operations, and type constructors using values (and not only types) as arguments.

1 Main Idea

The concept of multi-level algebra was initiated from our work on extending data models by new data types [2]. Although many-sorted algebra can be conveniently used to describe non-standard data models many important aspects remain unformalised. Even the generalisation to order-sorted algebra [4], though nicely expressing subtypes and the notions of inheritance and overloading, is not able to model fundamental concepts, such as, parametric polymorphism. Parametric order-sorted algebra [3] offers a partial solution, but there are still dependencies that cannot be expressed. For example, it is not clear, in general, how to define a parametric module that is not allowed to accept an instance of itself as parameter, which is needed, for instance, to define unnested sequences.

In contrast, this is possible with two levels of order-sorted algebra. The idea (in the two-level case) is to use a signature to describe a type system (or, language of types) where sorts denote sets of type names and operations denote type constructors. The values of an algebra for such a signature are then used as sorts of another signature now describing a language having the previously defined type system. This approach is not limited to two levels, and there are indeed reasonable applications of three-level algebras.

2 Kinds: Describing Ad Hoc and Subtype-Polymorphism

Suppose we have to define an operation "<" on numbers and strings (and possibly several other sorts). One approach is to give each signature entry separately. This becomes tedious as the number of data types for which "<" is defined grows. So it is much more convenient to group all the sorts in a *kind* [1], for example, ORD = {nat, int, str}, and then to define all signature entries by a type scheme:

$$\forall\, ord \in \mathrm{ORD}.\ {<}\colon ord \times ord \to \mathrm{bool}$$

Apart from saving space, this notation is more descriptive w.r.t. the language being defined since the ad hoc polymorphism of "<" is not "scattered" over different places in the signature.

Subtype-polymorphism, too, can be specified using kinds: First, define for each sort s (having subsorts) a kind SUB_s containing s and all subsorts of s. Then introduce for each operation $f\colon s \to t$ a specification:

$$\forall\, \sigma \in \mathrm{SUB}_s.\ f\colon \sigma \to t$$

3 Two-Level Algebra: Type Constructors and Parametric Polymorphism

A type constructor takes one or more types as arguments and produces a new type as result. The sequence constructor (seq), for example, takes a type, say, int, and produces the type containing all sequences of integers. Of course, seq may be applied to other types as well, but in some languages where nested sequences are not allowed (for instance, database languages) it must not be applied to sequence types. In that case, the argument types for seq are a proper subset of all types and can be grouped into an appropriate kind. Similarly, the result types form a kind, too.

Now we can regard kinds and type constructors as sorts and operations, respectively, of an order-sorted signature. The example of unnested sequences can then be expressed as:

*FernUniversität Hagen, Praktische Informatik IV, Postfach 940, 5800 Hagen, Germany, erwig@fernuni-hagen.de

typesystem UNNESTED
kinds ARG, SEQ
tcons int, str, bool: → ARG
  seq: ARG → SEQ

In the sequel we shall presume the variable quantifications "$\forall\ seq \in$ SEQ" and "$\forall\ arg \in$ ARG". Now we can specify operations on sequences as follows.

language LISTS
types from UNNESTED
funs nil: → *seq*
  cons: *arg* × seq(*arg*) → seq(*arg*)
  hd: seq(*arg*) → *arg*
  tl: seq(*arg*) → seq(*arg*)
  length: *seq* → int

Note that "$\forall\ seq \in$ SEQ. *seq*" denotes the same types as "$\forall\ arg \in$ ARG. seq(*arg*)". Thus, we can use *seq* in the type specifications for nil and length since we do not need to refer to the argument type of the respective sequences.

The signature UNNESTED defines merely the typing of type constructors. The semantics usually consists of two parts: On the one hand, algebraic properties of type constructors can be specified by equations (for instance, associativity of a product operator). The set of sorts is then taken modulo such a specification (in our example this was not necessary). On the other hand, the effects of type constructors on the carrier sets need to be given by additional functions. Formally, we can capture this by the following definition.

Definition (Multi-Level Algebra) An order-sorted signature is a 1^{st}-*level signature*, and an order-sorted algebra is a 1^{st}-*level algebra*. Given an n^{th}-level signature $(S, \leq, \Sigma)$ and an n^{th}-level Σ-algebra $\mathcal{B}$, an order sorted signature $(S', \leq', \Sigma')$ is an $n+1^{st}$-*level signature* depending on Σ and $\mathcal{B}$ if $S' = \bigcup_{s \in S} s^{\mathcal{B}}$. A Σ'-algebra $\mathcal{A}$ is an $n+1^{st}$-*level algebra* if for each $\sigma_{w,s} \in \Sigma$ there is a function $\sigma^{\mathcal{K}}_{w,s}$ (called *type constructor*) and if for each $s \in S'$ such that $s = \sigma^{\mathcal{B}}_{w,s}(t_1, \ldots, t_n)$ (with $w = s_1 \ldots s_n$ and $t_i \in s_i^{\mathcal{B}}$ for $1 \leq i \leq n$) we have $s^{\mathcal{A}} = \sigma^{\mathcal{K}}_{w,s}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$. The functions $\sigma^{\mathcal{K}}_{w,s}$ define the *constructor semantics* for Σ, and $\mathcal{A}$ *depends* on (the higher level) $\mathcal{B}$ and the constructor semantics for Σ. □

Note that the individual algebra levels are denoted by counting backwards (with regard to the construction history). That is, an $n+1^{st}$-level algebra $\mathcal{A}$ (or, $\mathcal{A}_1$) depending on the n^{th}-level algebra $\mathcal{B}$ (or, $\mathcal{A}_2$) is said to be on the first level whereas $\mathcal{B}$ is said to be on second level, and so on. In particular, when Σ is used to describe types, we also say that Σ is on *type level* and Σ' is on *language level*.

The constructor semantics for the seq constructor is defined by:

$$\text{seq}(s)^{\mathcal{A}} = \text{seq}^{\mathcal{K}}(s^{\mathcal{A}}) = (s^{\mathcal{A}})^*$$

4 Lifting

According to our definition, type constructors are working exclusively on types. But there are constructors that are also based on values. The array constructor, for example, takes in addition to its component type two values of an ordered type. Similarly, the string constructor takes a number n and denotes the set of strings of length n.

In order to retain the clear separation of the kind/type/value levels Cardelli [1] proposes to "lift" values onto the type level (and the corresponding types onto the kind level). With regard to the two examples, this means to introduce for each value $n \in$ nat a new type $\widehat{n}$ with the carrier being $\widehat{n}^{\mathcal{A}} = \{n\}$. Moreover, we create a new kind, $\widehat{\text{nat}}$, with $\widehat{\text{nat}}^{\mathcal{B}} = \{\widehat{n} \mid n \in \text{nat}^{\mathcal{A}}\}$. Then array and string can be used exclusively on the type level, as in array($\widehat{1}, \widehat{9}$, bool) or string($\widehat{23}$).

Let Σ_L denote the subset of type constructors that need lifted types. In order to specify a type system and language using types constructed by operations of Σ_L the following steps have to be performed (for a two-level algebra):

(i) Define the type system without Σ_L. Call the signature Σ_0.

(ii) Define Σ'_0, the part of the language not needing types constructed by Σ_L.

(iii) Perform lifting of Σ_0 and Σ'_0, and add Σ_L to $\widehat{\Sigma_0}$, that is, define $\Sigma = \widehat{\Sigma_0} \cup \Sigma_L$.

(iv) Finally, define Σ' with regard to Σ.

If there are constructors that use values of a type that is built by a constructor of Σ_L we have to repeat the last two steps. If only one lifting is necessary, we can specify Σ_L together with Σ_0 in one step. Thus, array can be defined by (we do not list lifted kinds explicitly):

typesystem ARRAYS
kinds ANY
tcons nat, str, bool: → ANY
    array: $\widehat{\text{nat}} \times \widehat{\text{nat}} \times \text{ANY} \to \text{ANY}$

Since sorts constructed by array are of kind ANY nested arrays are allowed by this definition — compare this to the definition of seq from above. (The same effect can be achieved by exploiting the poperties of order-sorted algebra and defining a kind ARR with ARR $\leq$ ANY.) The constructor semantics are given by:

$$\text{array}(\widehat{n},\widehat{m},t)^{\mathcal{A}} = \text{array}^{\mathcal{K}}(\widehat{n}^{\mathcal{A}},\widehat{m}^{\mathcal{A}},t^{\mathcal{A}}) = \text{array}^{\mathcal{K}}(\{n\},\{m\},t^{\mathcal{A}}) = \{n,\ldots,m\} \to t^{\mathcal{A}}$$

Operations on arrays can be defined by (assume quantifications "$\forall$ *any* $\in$ ANY" and "$\forall\, \widehat{n},\widehat{m} \in \widehat{\text{nat}}$"):

types from ARRAYS
funs newarray: nat × nat × *any* → array($\widehat{n},\widehat{m}$, *any*)
    select: array($\widehat{n},\widehat{m}$, *any*) × nat → *any*
    update: array($\widehat{n},\widehat{m}$, *any*) × nat × *any* → array($\widehat{n},\widehat{m}$, *any*)

Note that with the above definition range checking (for select/update) is not expressible on the type level, for example, an expression select(newarray(1, 9, true), 15) is type correct w.r.t. to the above signature. By introducing a third algebra-level range checking will become possible. (Then arrays can be defined in a more general fashion based on a class of subrange types.)

5 Three-Level Algebras

Consider the function [ ] for constructing sequences, which is defined for an arbitrary number of arguments. The signature entries are:

[ ]: → *seq*
[ ]: *ary* → seq(*ary*)
[ ]: *ary* × *ary* → seq(*ary*)
...

To denote these signature entries we need for each argument type t a kind containing all product types over t. This can be achieved as follows: We define a *kind constructor* list (this is an operation on level three with the same semantics as seq). Now, list(K) denotes for a kind K all sequences of sorts from K. If, for example, $K_{\text{nat}}{}^{\mathcal{S}} = \{\text{nat}\}$[1], the quantification "$\forall$ *ary* $\in$ list(K_{nat})" binds the sequences $\langle\rangle$, $\langle\text{nat}\rangle$, $\langle\text{nat},\text{nat}\rangle$, ... to *ary*. The desired product types can be obtained by "inserting" a "×" type constructor between each two adjacent types in a sort sequence. This is achieved by the higher order function fold:

$$\begin{aligned}
\text{fold}^{\mathcal{K}}(\sigma,\langle\rangle) &= \epsilon \\
\text{fold}^{\mathcal{K}}(\sigma,\langle t_1\rangle) &= t_1 \\
\text{fold}^{\mathcal{K}}(\sigma,\langle t_1,t_2,\ldots,t_n\rangle) &= \sigma(t_1,\text{fold}^{\mathcal{K}}(f,\langle t_2,\ldots,t_n\rangle))
\end{aligned}$$

Now the type of [ ] (for nat-sequences only) can be specified by:

[ ]: fold(×, list(K_{nat})) → seq(nat)

A more precise account of this kind of specification requires higher order algebras [8, 9] and a more elaborate treatment of lifting. Finally, for the convenient specification of multi-level algebras we need a language that allows for the use of terms of all levels in the definition of operations' ranks. This will be covered by the full paper.

6 Conclusions and Related Work

Data models are still an area of ongoing research. Some reasons for this may be the constant identification of new applications for database systems and the desire for improving existing models. All the more

[1] K_{nat} can be obtained by lifting.

it is surprising that no general framework is used, though, to describe the large variety of models. By using multi-level algebra we can describe different models within the same formalism. In the first place, this helps exhibiting relationships and differences which is necessary to fully understand and judge data models. Possibly, this could be used for implementations of one model by means of another or for investigations in the integration of heterogeneous database settings. If nothing else, the presented framework reveals the high complexity of seemingly simple data models, for example, the description of the relational model needs the full range of concepts indicated above (that is, lifting, three levels, higher order algebra).

In fact, two-level algebras were already used in [12] to specify categories with certain properties for theoretical investigation and in [7] for the formalization of the composition of specifications. In contrast, our concern is the specification of type systems, more specifically, the formal description of data models and query languages. In this respect, the work of [5] is similar, although more directed towards the description of a specific system architecture. Particular differences are that [5] does not consider lifting, that no specification language exists, and that the approach (like [12, 7]) is limited to two levels. Another difference between [12, 7] and our work is that we employ more than only one sort on level two. In [2] many applications of multi-level algebra can be found. This includes the formalization of graph types, heterogeneous sequences, and some operations with a variable number of arguments. In the full version of this paper we will give a specification of the relational model and an NF^2 model.

Finally, let us summarize some points counting in favor of using multi-level algebra and exhibiting its primary scope.

- Parametric polymorphism is expressible.
- All kinds of polymorphism (subtype, ad hoc, parametric) are describable within one formalism.
- Type systems can be easily extended by new structures (graphs, heterogeneous sequences). This is important to meet changing requirements of new applications.
- The definition of properties of type constructors (for example, associativity) is separated from the constructor semantics.
- Recently, fairly general approaches to the type checking of languages defined by many-level signatures have become available [10, 6, 11]. In many cases, these methods are directly applicable to languages defined by multi-level algebra.

References

[1] Cardelli, L.: Types for Data Oriented Languages, *Conf. on Extending Database Technology*, 1988, LNCS 303, pp. 1–15.

[2] Erwig, M., Güting, R.H.: Explicit Graphs in a Functional Model for Spatial Databases, Report 110, Fern-Universität Hagen, 1991.

[3] Goguen, J.A.: Higher-Order Functions Considered Unnecessary for Higher-Order Programming, in: David Turner (ed.) *Research Topics in Functional Programming*, Addison-Wesley, 1990, pp. 309–352.

[4] Goguen, J.A., Meseguer, J.: Order-Sorted Algebra I, Report SRI International, 1989.

[5] Güting, R.H.: Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization, *ACM SIGMOD Conf. on Management of Data*, 1993, to appear.

[6] Kaes, S.: Type Inference in the Presence of Overloading, Subtyping and Recursive Types, *ACM Conf. on Lisp and Functional Programming*, 1992, pp. 193–204.

[7] Leszczylowski, J., Wirsing, M.: Polymorphism, Parameterization and Typing: An Algebraic Specification Perspective, *Symp. on Theoretical Aspects of Computer Science*, 1991, LNCS 480, pp. 1–15.

[8] Meinke, K.: Universal Algebra in Higher Types, *7th Workshop on Specification of Abstract Data Types*, 1990, LNCS 534, pp. 185–203.

[9] Möller, B.: Algebraic Specifications with Higher-Order Operations, in: L.G.L.T. Meertens (ed.) *Program Specification and Transformation*, Elsevier Science Publishers, 1987, pp. 367–398.

[10] Nipkow, T., Prehofer, C.: Type Checking Type Classes, *20th ACM Symp. on Principles of Programming Languages*, 1993, pp. 409–418.

[11] Nipkow, T., Snelting, G.: Type Classes and Overloading Resolution via Order-Sorted Unification, *ACM Conf. on Functional Programming and Computer Architecture*, 1991, LNCS 523, pp. 1–14.

[12] Poigné, A.: On Specifications, Theories, and Models with Higher Types, *Information and Control 68*, 1986, pp. 1–46.

An Overview of the SODA System (Extended Abstract)

Peter Thiemann
Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, D-W7400 Tübingen, Germany

1 Introduction

We propose a system for software development which is aimed at merging the advantages of using methods from algebraic specification with features known from object-oriented systems, namely rapid prototyping, evolutionary programming, and reusability. Our proposal is a refinement of earlier work where we proposed to access functionally specified abstract data types from imperative modules [5].

A project is composed from modules with the usual operations of import, parametrization, and renaming. There are three kinds of modules. Modules can be either functional, state machine, or imperative.

Functional modules are specified in the executable first-order specification language SODA (specification in order-sorted data algebras). Data algebras are initial algebras of a modest extension of order-sorted algebra [2] by sort constructors and parametric polymorphism [6]. Derived functions are defined by recursive definitions as a conservative extension of the data algebra.

The other kinds of modules (state machine and imperative) describe the non-functional parts of a project (*e.g.*, interaction, database access). Their operations can not be used from functional modules. A state machine module defines a state and operations to manipulate the state and/or provide information about the state to the outside.

Imperative modules play an important rôle for our system to be interesting for real world projects. At any point of the design process the implementation of a functional or state machine module can be replaced by an imperative module. Of course there is the requirement that replacements of functional modules remain side effect free.

2 Foundation

The foundation of the data algebra for functional and state machine modules is an extension of order-sorted algebra (OSA). While ordinary OSA employs a partial order of sorts and provides parameterization only through module instantiations, we follow Hanus [3] by extending OSA with parametric polymorphic sort constructors and with a mechanism similar to record extensions as proposed by Wirth for the language Oberon [9, 10]. While Hanus defines a two-level semantics for polymorphic structures, we give a different (one-level) semantics as an algebra that employs truly polymorphic data values.

2.1 Preliminaries

$\wp^{fin}(M)$ denotes the set of finite subsets of M. For a function $f: A \rightarrow B$ and $A' \subseteq A$ $f|_{A'}$ denotes the restriction of f to A'. We write $\overline{\alpha_k}$ for the k-tuple $(\alpha_1, \ldots, \alpha_k)$.

A ranked alphabet Θ is a finite set of symbols with a total function $a: \Theta \rightarrow \mathbb{N}$ denoting the arity of the symbol, $\chi \in \Theta^{(k)}$ abbreviates $\chi \in \Theta$ and $a(\chi) = k$. The set $T_\Theta(V)$ of Θ-terms over a set of variables V is the smallest set T where $V \cup \Theta^{(0)} \subseteq T$ and for all $k \in \mathbb{N}$, $\chi \in \Theta^{(k)}$, $t_1, \ldots, t_k \in T$ it holds that $\chi(t_1, \ldots, t_k) \in T$. If $V = \emptyset$ we write T_Θ. The set of all variables occurring in term t is var(t). A substitution is a function $\sigma: V \rightarrow T_\Theta(V)$ where $\sigma v \neq v$ only for finitely many $v \in V$. Denote the set of substitutions over $T_\Theta(V)$ by Subst(Θ, V). A substitution σ is extended to a function $\hat{\sigma}: T_\Theta(TV) \rightarrow T_\Theta(TV)$ by $\hat{\sigma}|_V = \sigma$, $\hat{\sigma}|_{\Theta^{(0)}} = id$, and for $\chi \in \Theta^{(k)}$ and $t_1, \ldots, t_k \in T_\Theta(TV)$ $\hat{\sigma}(\chi(t_1, \ldots, t_k)) = \chi(\hat{\sigma}t_1, \ldots, \hat{\sigma}t_k)$. For convenience we write σ instead of $\hat{\sigma}$. A renaming is a substitution that permutes the variables. Renamings induce an equivalence relation $\equiv \subseteq T_\Theta(V)^2$.

2.2 Sorts and signatures

A polymorphic order-sorted signature $\Sigma = (\Theta, \Delta)$ consists of a ranked alphabet Θ of *sort constructors* and a finite set Δ of *operator symbols* with a total function $a: \Delta \rightarrow \wp^{fin}(\bigcup_{n \in \mathbb{N}} D_n) \setminus \{\emptyset\}$ where $D_n = \{(\tau_1 \ldots \tau_n, \tau_0, C) \mid \tau_i \in T_\Theta(TV), C \in \mathcal{C}\}$ denoting the arity of the symbol. $\mathcal{C}$ is the set of constraint sets: $\mathcal{C} = \wp^{fin}(TV \times T_\Theta(TV))$; if $C \in \mathcal{C}$ and $\alpha \in TV$ there is at most one pair of the form $(\alpha, \tau) \in C$, furthermore C can be linearly ordered to $\{(\alpha_1, \tau_1), \ldots, (\alpha_m, \tau_m)\}$ so that α_i occurs in τ_j only if $i < j$.

A polymorphic data structure declaration (PDSD) simultaneously defines a relation $\leq$ on T_Θ, sort terms without variables, and the arity of the data constructors. A PDSD is a system of equations of the form $\chi(\overline{\alpha_l}) = \ldots + \tau + \ldots + (\ldots, cd, \ldots)$ where $\chi \in \Theta^{(l)}$, $\alpha_i \in TV$, and $\tau, \tau_i \in T_\Theta(\{\alpha_1, \ldots, \alpha_l\})$. A *constructor declaration* $cd ::= c \mid c(\overline{\tau_k})$ declares the arity of $c \in \Delta$ to be $a(c) = \{(\tau_1 \ldots \tau_k, \chi(\overline{\alpha_l}), \emptyset)\}$.

The sort graph of a PDSD is the directed graph with vertices Θ and edges $(\chi \rightarrow \chi')$ if there is an equation $\chi(\overline{\alpha_l}) = \chi'(\overline{\tau_{l'}}) + \ldots$. Call a PDSD *well-formed* if its sort graph is acyclic.

For a well-formed PDSD define a rewrite relation $\succ \subseteq T_\Theta(TV)^2$ by $\chi(\tau_1, \ldots, \tau_l) \succ \tau'$ if there is an equation $\chi(\overline{\alpha_l}) = \tau + \ldots$ and $\tau' = \tau[\tau_i/\alpha_i]$. $\overset{*c}{\succ}$ denotes the reflexive, transitive, and Θ-compatible closure of $\succ$.

Define $\leq \subseteq T_\Theta \times T_\Theta$ by $t_1 \leq t_2$ if either $t_1 = \chi(\tau_1, \ldots, \tau_k)$, $t_2 = \chi(\tau'_1, \ldots, \tau'_k)$ and $\tau_i \leq \tau'_i$ for $1 \leq i \leq k$, or $t_2 \succ t'_2$ and $t_1 \leq t'_2$. The relation $\leq$ is a partial order. $\leq$ can be extended to $T_\Theta(TV)$ by adding the rules $\alpha \leq \alpha$ for all $\alpha \in TV$ and defining $t_1 \leq' t_2$ if $\exists t'_2 \equiv t_2$ such that $t_1 \leq t'_2$.

Call a well-formed PDSD *coherent* if $\leq$ is a type order, *i.e.*, $\leq$ is a partial order and if there is an upper (lower) bound of $\tau_1, \tau_2 \in T_\Theta(TV)$ then there is a least upper (greatest lower) bound denoted by $\tau_1 \sqcup \tau_2$ $(\tau_1 \sqcap \tau_2)$. For technical reasons we require all PDSDs to be coherent.

2.3 Algebras

A polymorphic order-sorted algebra (A, ι) with signature $\Sigma = (\Theta, \Delta)$ consists of

- a family $A = \{A^\tau \mid \tau \in T_\Theta(TV)\}$ of carrier sets indexed by (equivalence classes of) sort terms where for all $\tau, \tau' \in T_\Theta(TV)$ $A^\tau \subseteq A^{\tau'}$ if $\tau \leq \tau'$ and also for all $\sigma \in \mathrm{Subst}(\Theta, TV)$ it holds that $A^\tau \subseteq A^{\sigma\tau}$, and
- a total function $\iota: \Delta \rightarrow \mathrm{Ops}(A)$ (an interpretation) where $\mathrm{Ops}(A) = \bigcup_{n \in \mathbb{N}} \{f: A^{\tau_1} \times \ldots \times A^{\tau_n} \rightarrow A^{\tau_0} \mid \tau_i \in T_\Theta(TV)\}$ and ι maps $f: (\tau_1 \ldots \tau_n, \tau_0, C) \in \Delta$ to an element of $\bigcap \{A^{\sigma\tau'_1} \times \ldots \times A^{\sigma\tau'_n} \rightarrow A^{\sigma\tau_0} \mid \sigma \in \mathrm{Subst}(\Theta, TV), \alpha \leq \tau \in C \Rightarrow \sigma\alpha \leq \tau, \tau_i \overset{*c}{\succ} \tau'_i\}$.

2.4 Terms

During the formation of terms we are given a value v of sort τ and want to apply operation $f: (\tau_1, \tau_0, C_f) \in \Delta$. f is applicable to v if there is a substitution σ such that $\sigma\tau \leq \sigma\tau_1$ and the inequations σC_f are satisfied. We give a non-deterministic procedure SOLVE that is an adaption of the algorithm MATCH of [1] to our situation combined with some simplification rules. The procedure is given a set C_0 of inequations on sort terms as input. Upon termination it either yields a substitution σ that satisfies C_0 or fails if no such σ exists. The following deduction rules are applied to the initial set of inequations $C_f \cup \{\tau \leq \tau_1\}$ and the identity substitution *id*.

$$\frac{C \cup \{\chi(\overline{\tau_l}) \leq \chi(\overline{\tau'_l})\}, \sigma}{C \cup \{\tau_1 \leq \tau'_1, \ldots, \tau_l \leq \tau'_l\}, \sigma} \qquad (2.1) \qquad\qquad \frac{C \cup \{\tau \leq \tau'\}, \sigma \quad \tau \notin TV}{C \cup \{\tau \leq \tau''\}, \sigma} \quad \tau' \succ \tau'' \qquad (2.2)$$

$$\frac{C \cup \{\alpha \leq \alpha\}, \sigma \quad \alpha \in TV}{C, \sigma} \qquad (2.3) \qquad\qquad \frac{C \cup \{\alpha \leq \beta\}, \sigma \quad \alpha \neq \beta \in TV}{C[\alpha/\beta], \sigma \circ [\beta \mapsto \alpha]} \qquad (2.4)$$

$$\frac{C \cup \{\alpha \leq \tau, \alpha \leq \tau'\}, \sigma}{C \cup \{\alpha \leq \tau \sqcap \tau'\}, \sigma} \quad \text{fail, if } \not\exists \tau \sqcap \tau' \qquad (2.5) \qquad\qquad \frac{C \cup \{\tau \leq \alpha, \tau' \leq \alpha\}, \sigma}{C \cup \{\tau \sqcup \tau' \leq \alpha\}, \sigma} \quad \text{fail, if } \not\exists \tau \sqcup \tau' \qquad (2.6)$$

$$\frac{C \cup \{\tau \leq \alpha, \alpha \leq \tau'\}, \sigma}{C \cup \{\tau \leq \tau', \tau \leq \alpha\}, \sigma} \quad \alpha \notin \mathrm{var}(\tau) \cup \mathrm{var}(\tau') \quad (2.7)$$

$$\frac{C \cup \{\tau \leq \alpha\}, \sigma}{C[\tau/\alpha], \sigma \circ [\alpha \mapsto \tau]} \quad \alpha \notin \mathrm{var}(\tau) \qquad (2.8) \qquad\qquad \frac{C \cup \{\alpha \leq \tau\}, \sigma}{C[\tau/\alpha], \sigma \circ [\alpha \mapsto \tau]} \quad \alpha \notin \mathrm{var}(\tau) \qquad (2.9)$$

The rules are applied according to the following plan. First rule (2.1) is applied as often as possible. If now rule (2.2) is applicable we apply it by non-deterministically choosing a rewrite step and fall back to rule (2.1). If rule (2.2) was not applicable but there is still an inequation $\tau \leq \tau'$ where τ and $\tau' \notin TV$ the procedure signals failure and backtracks to the last application of rule (2.2) where there is an alternative left. Otherwise rules (2.3) and (2.4) are repeatedly applied. Now rules (2.5) and (2.6) are iterated. Failure at these rules is not handled with backtracking. Now all inequations have the form $\alpha \leq \tau$ or $\tau \leq \alpha$ and for each variable α there is at most one inequation $\alpha \leq \tau$ and at most one inequation $\tau \leq \alpha$. Rule (2.7) deals with the case that both inequations are present for a given variable α. After its application the procedure must fall back to rule (2.1).

At this stage we have at most one inequation for every variable and it either has the form $\alpha \leq \tau$ or $\tau \leq \alpha$. The resulting substitution is built with rules (2.8) and (2.9).

The procedure SOLVE has type $\wp^{fin}(T_\Theta(TV)^2) \rightarrow \mathrm{Subst}(\Theta, TV)$. On input C it runs the rule system as outlined above on the initial configuration C, id. If the rules terminate successfully with configuration $\emptyset, \sigma$ then σ is output, otherwise failure is signalled.

For a signature $\Sigma = (\Theta, \Delta)$ define the *term sets* T^τ for $\tau \in T_\Theta(TV)$ as the smallest solution of

- if $f\colon (\varepsilon, \tau_0, \emptyset) \in \Delta$ then $f \in T^{\tau_0}$,
- if $f\colon (\tau_1 \ldots \tau_n, \tau_0, C) \in \Delta$ and $t_i \in T^{\tau_i'}$ for $1 \leq i \leq n$ and $\sigma = \mathrm{SOLVE}(\{\tau_i' \leq \tau_i \mid 1 \leq i \leq n\} \cup C)$ then $f(t_1, \ldots, t_n) \in T^{\sigma\tau_0}$. $(\mathrm{var}(\tau_i) \cap \mathrm{var}(\tau_i') = \emptyset$ can be assumed.)

The *term algebra* is defined as (T, ι) where the carrier $T = \{\overline{T}^\tau \mid \tau \in T_\Theta(TV)\}$ and $\overline{T}^\tau = \bigcup\{T^{\tau'} \mid \tau' \overset{oc}{\succeq} \tau'' \wedge \exists \sigma \in \mathrm{Subst}(\Theta, TV) : \sigma\tau'' = \tau\}$. The interpretation ι is defined for $f\colon (\varepsilon, \tau, \emptyset) \in \Delta$ by $\iota(f) = f$ and for $f\colon (\tau_1 \ldots \tau_n, \tau_0, C)$ and $t_i \in \overline{T}^{\tau_i'}$ by $\iota(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ if $\mathrm{SOLVE}(\{\tau_i' \leq \tau_i \mid 1 \leq i \leq n\} \cup C)$ succeeds.

At this stage homomorphisms can be defined and the initiality of the term algebra can be proved. A semantics for the derived functions can be obtained in several ways. Either the principle of structural induction is used to define total functions on initial polymorphic OSAs (as proposed by Klaeren for many sorted algebras [4]), or monotone algebras are used to give a fixpoint semantics for general recursive functions.

3 Methodology

In the following subsections we briefly review the specification and programming facilities that may be used in the different kinds of modules. In order not to lose referential transparency we do not allow arbitrary calls between modules: imperative modules may call any function that is desired, state machine modules cannot call functions from imperative modules, and functional modules must only access functions from functional modules. This concept (confined to functional and imperative modules) originates from [5].

3.1 Functional modules

Functional modules are composed of an interface, a local declarations, and a function definitions. The interface provides means to import and combine signatures and sort structures from other modules. Parts of imported signatures can be projected and renamed. Imported and locally declared entities can be declared visible for export. Nothing is exported by default. The local declarations extend the combined imported signatures by providing PDSDs as described in 2.2 and declaring operator symbols. Function definitions are mutually recursive definitions of locally declared operator symbols. Let $\Sigma = (\Theta, \Delta)$ denote the signature formed by imports and local declarations. The form of a definition is $f(x_1, \ldots, x_n) = e$ where $f\colon (\tau_1 \ldots \tau_, \tau_0, C) \in \Delta$ is locally declared and $e \in E_\Sigma(\{x_1, \ldots, x_n\})^{\tau_0, C}$ under the assumption that $x_i \in E^{\tau_i, C}$. The set of right hand side expressions E_Σ is an extension of T_Σ by the constructions **let** $v = e$ **in** e' and **case** e_0 **of** $\ldots\ c_j(v_1, \ldots, v_k) : e_j\ \ldots$.

Modules can be parameterized with respect to a signature. A parameter signature is specified by declarations of operator symbols and by PDSDs without constructor declarations. Parameter instantiation is provided as an extension to the import facility in the interface section. Proper instantiation is checked by matching signatures.

3.2 State machine modules

A state machine module can be regarded as the definition of an object class. It has interface and local declarations in the same way as functional modules. Additionally it declares a specific sort to be the state. The state sort can also be an extension of a supersort. This means that the arities of all constructors of the supersort are extended with identical additional components. Thus we can have multiple inheritance statically by means of OSA and single inheritance dynamically by means of the constructor extensions. State machine entities can be created dynamically and each entity has its own local state which is initialized at creation time.

Furthermore, a state machine module contains definitions of operations that take the state as implicit argument and may update it destructively. Operations may invoke other operations, and they can return values.

An operation f is defined by $f(x_1, \ldots, x_n) = \mathtt{let}\ m_1; \ldots; m_l\ \mathtt{return}\ e$, $e \in E_\Sigma$, where each $m_i \in M_\Sigma$, *i.e.*, it is either an expression $m \equiv e \in E_\Sigma$, or an assignment $m \equiv v = e$ for $e \in E_\Sigma$, or a case decomposition $m \equiv \mathbf{case}\ e\ \mathbf{of}\ \ldots c_j(v_1, \ldots, v_k) : \overline{m} \ldots, \overline{m} \in M_\Sigma^*$.

3.3 Imperative modules

In imperative modules we use the algebraic data structures as a type system for an ordinary imperative language. The programming language Oberon already has a record extension mechanism similar to our proposed constructor extensions.

4 Conclusion

The concept outlined above appears promising since it combines a specification method for abstract data types with clean handling of state. The restrictions that we impose on inter-module calls allow for an efficient implementation since it is possible to take advantage of the referential transparency in the implementation of functional modules. Most of these advantages carry over to state machine modules since side effects are restricted to updating the state. The imperative modules are provided as a last resort, for example for system level operations.

By using an order-sorted framework we gain flexibility compared to our earlier work which builds on many-sorted algebras and does not have the concept of state machine modules [5]. The improved flexibility entails better reusability and the possibility for evolutionary program design.

We have a functional language implementation toolkit around an implementation technique that we have developed in earlier work [8, 7]. We are currently preparing an implementation of the front end for the functional part of SODA in this environment. Furthermore we investigate the extensions needed to implement the state machine modules. For the imperative part we consider an extension of the programming language Oberon [9] with algebraic datatypes and overloading.

References

[1] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In H. Ganzinger, editor, *ESOP 1988*, pages 94–114, 1988. LNCS 300.

[2] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, Menlo Park, CA, July 1987.

[3] M. Hanus. Parametric order-sorted types in logic programming. In *Proc. TAPSOFT 1991*, pages 181–200, 1991. LNCS394.

[4] H. Klaeren. A constructive method for abstract algebraic software specification. *Theoretical Comput. Sci.*, 30:139–204, 1984.

[5] H. Klaeren and P. Thiemann. A clean Modula-2 interface to abstract data types. *Structured Programming*, 11:69–77, Apr. 1990.

[6] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[7] P. Thiemann. LaToKi: A language toolkit for bottom-up evaluation of functional programs. In M. Bruynooghe and M. Wirsing, editors, *Proc. PLILP '92*, page 481f, Leuven Belgium, Aug. 1992. Springer. LNCS 631.

[8] P. Thiemann. Efficient implementation of structural recursive programs. *Journal of Programming Languages*, 1(1):41–70, Mar. 1993.

[9] N. Wirth. The programming language Oberon. *Software, Practice and Experience*, 18(7):671–690, July 1988.

[10] N. Wirth. Type extensions. *ACM Trans. Prog. Lang. Syst.*, 10(2):204–214, Apr. 1988.

Category Theory for the Configuration of Complex Systems

Gillian Hill *

The abstract framework of category theory is shown to provide a precise semantics for the configuration of complex systems from their component parts. Diagrams, defined as functors between categories, express configuration by representing the operations of combinators on recursively defined system components. Although modularity has been described as an essential property of complex systems, no clear and simple definition of a module has emerged at this general level. In this paper a new module concept is defined to represent reusable system components, at any level of development. The semantics of system configuration is given by the construction of colimit diagrams.

A category theoretic semantics was given in [BG77] for putting theories together to make specifications. The activity of specification was viewed as theory-building in [TM87, Vel85] and interpretation between theories was formalized in a categorial framework in [MF90]. Category theory was used to define an abstract specification theory for refining specifications in [UG90]. In this paper these ideas are extended to provide a precise semantics for both structuring and implementing system components to configure a final executable system.

A language for configuration, designed in [Hil92], is at a meta-level to a specification language and describes the operations of combinators on the specifications and modules that represent the system components. High-level combinators express the horizontal structuring of both specifications and modules by extension and parameterization; and also the vertical development, which is part of the design process, by implementation. The relationships between the component parts of systems have been identified at an intuitive level in order to choose appropriate high-level combinators for configuration. The high-level combinators have then been defined formally in terms of the more primitive combinators: interpretation, extension and conservative extension. A logical system includes the configuration theory and the application theories, to be configured. The logic for configuration must possess the strong Craig interpolation property in order to preserve conservative extensions of structured specifications under interpretation.

The aim of this paper is to present a logical framework for the configuration of modular systems that is independent of any particular specification approach, design methodology or programming paradigm. First-order logic is chosen to express new operations for horizontal structuring and vertical implementation within a conceptual framework that is both simple and natural for engineers to use. System components are represented in a uniform development space by recursively defined objects with sorts in the set {*specification, module*}. In [Hil92] specifications are presented as objects in textual form that record the history of configuration as a sequence of operations by the combinators on recursively defined objects. In this paper the abstract category-theoretic semantics is functional with the primitive combinators represented by natural transformations between recursively defined diagrams.

We define a diagram as a functor from the category of graphs to the category **Conf** of configured objects when the limit of the functor exists in **Conf**. The functor labels a graph, which has a only a shape, by the specifications or modules, as the objects, at the nodes and by the morphisms between the objects as the arrows. The natural transformations that join objects, represented by diagrams, to form more structured diagrams become the morphisms between objects in the diagrams of the more structured objects. The semantics of the high-level combinators is given by the construction of colimit diagrams that express the joining together of structured objects that may share common parts. Morphisms representing conservative extensions are shown to be crucial for completing the construction of these colimit diagrams. Our definition of a diagram is based on that in [GMU88] but we change their presentation of the structured specifications. In addition we provide a more concrete semantics for our configuration language. This is given by a set of well-formed diagrams, that name both the objects at the nodes and the primitive combinators

*Department of Computing, Imperial College of Science Technology, London, SW7 2BZ, gah@doc.ic.ac.uk
and: Department of Computer Science, City University, London EC1 V0HB, gah@cs.city.ac.uk

that operate on these objects. These more concrete diagrams are the elements in the models for the configuration language and represent the structured objects.

Our new module concept is proposed as an aid to managing the complexity of large systems by focusing on building systems from reusable components at any stage in system development. A module is created by a combinator from the textual specification of an object, and the semantics of module creation is based on the construction of a colimit for the diagram of that structured specification. Any number of uniquely named module instances can be created from a specification at any stage in system configuration. Modules may be created from primitive specifications before they are structured, or alternatively from complex specifications at the end of the structuring process. Similarly modules may be created from abstract specifications before they are implemented, or alternatively they may be created from concrete specifications at the end of the refinement process.

As a simple example, **obj** one_roomed_house, structured as **spec** house[**module** room1], may be instantiated to **spec** house $_{\text{module room1}}$ [**module** bedroom]. Alternatively, **obj** two_roomed_bungalow may be instantiated to **spec** house $_{\text{module room1,module room2}}$ [**module** kitchen, **module** bedroom]. Module instances, uniquely named, may then be created for each of these structured specifications by an operation which is safer than low-level copying. The module instance of the house with one room as a bedroom will be structurally identical to the module instance of **spec** house $_{\text{spec room}}$ [**spec** bedroom]. Their textual specifications would record different histories of specification, however: the first with instantiation by a module; the second by a specification. In the abstract semantics instantiation by a module is represented by interpretation between diagrams whose single nodes are colimit objects; instantiation by a specification is by interpretation between structured diagrams. To configure a house with three living rooms and three bedrooms we would structure a house parameterized by two types of room, **spec** house[**spec** room1, **spec** room2]. Three modules created from each of the specifications for a room could then be instantiated to the actual room required, such as lounge or guest bedroom.

Our approach is intended to be loose and flexible. The engineer is able to choose, at each stage in building a system, between building from specifications or modules. The final configured object of a software system will be a structured object that is implemented and in the form of an executable module. In addition to this flexibility the engineer is able to express explicitly the sharing or non-sharing of system components. Flexibility is also provided within our theory of configuration by the commutative properties of the high-level combinators. We characterize these properties as the axioms for our theory of configuration in the style of the algebraic calculus of [BHK90].

References

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceedings of the 5th. International Joint Conference on Artificial Intelligence*, pages 1045–1058, Cambridge, Mass., 1977.

[BHK90] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, April 1990.

[GMU88] T. Gergely, T. Maibaum, and L. Ury. Modularization: Structuring specifications. Technical Report V1.0, Applied Logic Laboratory, SZÁMALK, Budapest and Imperial College, November 1988.

[Hil92] Gillian Hill. *A Language for System Configuration*. PhD thesis, Department of Computing, Imperial College, University of London, 1992. draft.

[MF90] T. Maibaum and J. Fiadeiro. Stepwise program development in Π institutions. Technical report, Imperial College, November 1990. with Martin Sadler.

[TM87] Wladyslaw M. Turski and Thomas S. E. Maibaum. *The Specification of Computer Programs*. International Computer Science Series. Addison Wesley, 1987.

[UG90] László Úry and Tamás Gergely. A constructive specification theory. In G. David, R. T. Boute, and B. D. Shriver, editors, *Declarative Systems*. Elsevier Science Publishers B. V. (North Holland) IFIP, 1990.

[Vel85] Paulo A. S. Veloso. Program development as theory manipulations. Technical report, PUC/RJ Departamento de Informatica, Rio de Janeiro, Brazil, May 1985. Series: Monografias em Ciencia da Computacao No4/85.

Algebraic-Oriented Institutions
Extended Abstract*

M. Cerioli and G. Reggio.
Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova
Viale Benedetto XV, 3 - 16132 Genova - Italy
e-mail: {cerioli, reggio}@disi.unige.it

In many recent applications of the algebraic paradigm to formal specification methodologies, already known algebraic frameworks (one-sorted/many-sorted/order-sorted total/partial/non-strict/generalized algebras with/without predicates equipped by equational/conditional/Horn-Clauses/first-order logic) are endowed with features, tailored for special purposes, in order to improve software development; for example think of higher-order functions and, in the field of concurrency, dynamic elements, temporal logic combinators, or event logic combinators (see e.g. [1, 4, 7, 8, 9, 10]).

Although it is often the case that the new features are in a sense orthogonal to the underlying algebraic framework and that the same construction applies to any sufficiently expressive formalism, in the practice *ad hoc* theories are developed, neglecting the parametric definition such theories are instances of. This lack of generality is conflicting with the ability of changing the basic formalism, and hence with the reuse of methodologies, seen as high-level theoretical tools for the software development.

In any real application two steps can be distinguished in the process of getting the most suitable algebraic formalism: the choice of the most appropriate basic algebraic formalism (i.e. sufficiently powerful for the problem, but non-overcomplex) and the addition of the features needed in the particular case (e.g. entities for structured parallelism or higher-order functions for functional programming). Thus here we propose a modular construction of algebraic frameworks, formalized by means of operations on institutions, used as a synonym for logical formalism, in order to build richer institutions by adding one feature at a time.

Many constructions used in the practice have meaning only for those institutions that represent "algebraic formalisms". In order to give sound foundations for the treatment of such operations, a preliminary step is the formal definition of which institutions correspond to algebraic frameworks. Here we propose a first attempt at the definition of *algebraic-oriented* institutions, that includes all interesting cases.

Technically algebraic-oriented institutions are described by (standard) algebraic specifications, so that both theoretical and software tools are at hand to help in the building process; moreover algebraic specification users already have the know-how to understand and manipulate metaoperations building algebraic formalisms.

Using this definition of algebraic-oriented institutions, we formally define some operations adding features to basic algebraic frameworks and show that

the result of such operations applied to any algebraic-oriented institution is an algebraic-oriented institution, too; so that the result can be used as input for other

*This work has been partially supported by Esprit-BRA W.G. n.6112 Compass, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R. (Italy), MURST-40% Modelli e Specifiche di Sistemi Concorrenti operations, building in such way a formalism as rich as needed by the application.

What is an algebraic-oriented institution Let us give some hints of the basic characteristics of algebraic-oriented institutions.

Analyzing the algebraic formalisms present in literature it appears that signatures consist in all frameworks basically of a (possibly singleton) set of *sorts*, a (possibly empty) family of typed functions and a (possibly empty) family of typed predicates. Then these ingredients can be structured by means of (meta)functions, like higher-order functional sort or product sort constructors and accordingly built-in projection functions, or (meta)predicates, like the subsort relation for order-sorted signatures, or the observability property on sorts. This leads to characterize the *algebraic-oriented* signatures as subcategories of models of any partial algebraic specification including a standard part, sketched in the sequel, consisting of the (meta)sorts S, F and P together with the obvious *arity* total (meta)functions and the auxiliary subspecification *Streams*(**sorts** S) defining the sort *S_Stream* of streams on S.

```
spec $T_{sign}$ =
    enrich $Streams$(sorts $S$) by
    sorts $F, P$
    opns
        $arity: F \to S\_Stream$
        $P\_arity: P \to S\_Stream$
        $result\_type: F \to S$
    axioms
        $total(arity)$
        $total(P\_arity)$
        $total(result\_type)$
```

Thus, using this powerful internalization, the usual algebraic machinery is at hand in order to modularly define the requirements on the syntax, even using rapid prototyping tools.

Consider as an example of application the definition of order-sorted signatures, see e.g. [6], where the standard part is enriched by an extra sort to denote the names of functions, so that overloading can be allowed, keeping distinct the operations from the names used in the language to denote them, and a binary predicate of *subsort* on sorts. Axioms to make the subsort relation a partial order are imposed, too.

```
spec sign$_{OSA}$ =
    enrich $T_{sign}$ by
    sorts $Op\_Names$
    opns
        $name: F \to Op\_Names$
    preds
        $\_ \leq \_ : S \times S$
        $\_ \leq^* \_ : S\_Stream \times S\_Stream$
    axioms
        $total(name)$
        $s \leq s$
        $s \leq s' \wedge s' \leq s \supset s = s'$
        $s \leq s' \wedge s' \leq s'' \supset s \leq s''$
        $\Lambda \leq^* \Lambda$
        $w \leq^* w' \wedge s \leq s' \supset s \cdot w \leq^* s' \cdot w'$
        $name(f) = name(g) \wedge arity(f) \leq^* arity(g) \supset result\_type(f) \leq result\_type(g)$
    -   overloding preserves subsorting
        ⋮
```

These axioms define the minimal requirements that any order-sorted signature should satisfy; note that more sofisticated restrictions can be imposed as well, like *regularity* and *coherency*, using first-order axioms. Note also that morphisms between the models of this specification coincide with order-sorted signature morphisms.

Accordingly with signatures, models and sentences have to be restricted. Roughly speaking models should be some kind of algebras, i.e. they should interpret the elements of sort S by sets, the elements of sort F by (possibly partial or non-strict) functions with the correct arity and the elements of sort P by (possibly non-strict) predicates. In particular all "algebras" used in practice, like many-sorted, partial, order-sorted, non-strict, etc., can be easily seen in the above way.

Analogously to the choice made for signatures, sentences are defined as term algebras on uniform enrichments of their signatures by the *Var*, *Term*, *Atom* and *Sen* (meta)sorts and the obvious constructors for such sorts, together with the needed connectives, depending on the application that is intended to be faced, like first-order operators, temporal logic operators and so on.

An example of operation on algebraic-oriented institutions As an example of the operations supported by algebraic-oriented institutions, let us consider the introduction of elementary features for handling concurrency in any algebraic formalism. The intuitive idea is that some sorts classify *dynamic* elements and hence for any of those sorts a labelled transition system is introduced. Using the algebraic-oriented framework, this can be formalized by an operation dyn that on an algebraic-oriented institution $AO = (\mathbf{AOSign}, AOSen, AOMod, \models^{AO})$, whose signature category **AOSign** is the model category of a (partial) specification $T_{\mathbf{AOSign}}$ enriching $T_{\mathbf{sign}}$, gives as output the algebraic-oriented institution dyn(AO), whose signature category dyn(**AOSign**) is the model category of the following (partial) specification:

```
spec T_dyn(AOSign) =
    enrich  T_AOSign  by
    opns    label: S → S
            trans: S → P
    preds   Dyn: S
    axioms  P_arity(trans(s)) = s · label(s) · s
            Dyn(s) ⇔ D(trans(s))
    –       trans is defined only for dynamic sorts
            Dyn(s) ⇔ D(label(s))
    –       label is defined only for dynamic sorts
```

As the signature category dyn(**AOSign**) is a (full) subcategory of **AOSign**, models and sentences for dyn(AO) are simply the restrictions respectively of *AOMod* and *AOSen* to dyn(**AOSign**).

Several instances of this construction have been independently developed in applicative projects starting from different basic algebraic formalisms (see e.g. [1, 2]). Note that, as dyn(AO) is an algebraic-oriented institution too, it can be used as argument for further operations, adding a feature at a time, e.g. temporal logic combinators, in a modular way.

Relationships with other approaches This work continues and adds to [3], where some operations on institutions were proposed in order to deal with some uniform enrichment of logical formalisms, that, although arisen in the field of concurrency, have a general character

and can be defined on any institution (and on algebraic-oriented institutions result in algebraic-oriented institutions, too).

Algebraic-oriented institutions differ from the abstract algebraic institutions by Tarlecki (see e.g. [11]) not only in purposes, since algebraic-oriented institutions are designed to support the definition of operations among several institutions more than constructions on the models of one institution, but also from the technical point of view. Indeed using only the categorical characteristics of institutions (as in [11]) we cannot add the features of interest, since they explicitly involve the components of syntax and the elements of the algebraic models.

Although algebraic-oriented institutions share with parchments (see [5]) the intuition of using usual algebraic machinery to deal with institution ingredients, the aim of parchments is to define institutions starting from some basic syntactic elements in a canonical way.

References

[1] E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In *Proc. TAPSOFT'87, Vol. 1*, number 249 in L.N.C.S., Berlin, 1987. Springer Verlag.

[2] E. Astesiano and G. Reggio. A structural approach to the formal modelization and specification of concurrent systems. Technical Report PDISI-92-01, DISI, Università di Genova, Italy, 1992.

[3] M. Cerioli and G. Reggio. Institutions for very abstract specifications. Submitted, 1992.

[4] G. Costa and G. Reggio. Abstract dynamic data types: a temporal logic approach. In *Proc. MFCS'91*, number 520 in L.N.C.S., Berlin, 1991. Springer Verlag.

[5] J. Goguen and R. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charter and parchments. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydehard, editors, *Proceedings of Summer Workshop on Category Theory and Computer Programming*, number 240 in L.N.C.S., pages 313–333, Berlin, 1986. Springer Verlag.

[6] J. Goguen and R. Diaconescu. A survey of order sorted algebra. Draft, 1992.

[7] K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100(2), 1992.

[8] B. Möller, A. Tarlecki, and M. Wirsing. Algebraic specification with built-in domain constructions. In *Proc. of CAAP'88*, number 299 in L.N.C.S., Berlin, 1988. Springer Verlag.

[9] G. Reggio. Entities: an istitution for dynamic systems. In *Recent Trends in Data Type Specification*, number 534 in L.N.C.S., Berlin, 1991. Springer Verlag.

[10] G. Reggio. Event logic for specifying abstract dynamic data types. In *Recent Trends in Data Type Specification*, number 655 in L.N.C.S., Berlin, 1992. Springer Verlag.

[11] A. Tarlecki. Quasi-varieties in abstract algebraic institutions. *J. of Comp. and Syst. Science*, 33, 1986.

On the Correctness of Modular Systems

(Extended Abstract)

Marisa Navarro[†]
Fernando Orejas[*]
Ana Sánchez[†]

In the design and implementation of a large system modularity is a critical issue. Large systems need to be divided into blocks so that system development becomes more manageable, clear, modifiable and reusable. These blocks, known as modules, are self-contained entities with individual meaning that are connected among them in such a way that their interconnections define the intended software system. System design involves both the construction of the module structure at the specification level and the implementation of each module. Since implementation only appears at the level of each module, it would be desirable to ensure that the correct implementation of each module should guarantee the correct implementation of the whole software system.

In this paper, we study the correctness of modular systems in a simple framework, including both specification design and implementation. This framework may be described as follows:

We consider two institutions, SPEC and PROG, underlying the specification and programming languages, respectively. For simplicity, we assume that both institutions share the same category of signatures and the same model functor, i.e. they differ in the Sent functor and on the satisfaction relation. Additionally we assume that both institutions are semiexact, i.e. they have pushouts and amalgamations [EBCO 91], and are equipped with an inclusion system [DGS 91].

A *module* M is assumed to be a pair of specifications, M=(IMP,EXP), with IMP$\subseteq$EXP, where IMP and EXP denote, respectively, the import and export specifications of M. The results obtained can be generalized to more complex forms of modules, such as [EM 89, ST 89]. Its meaning is given by an associated constructor κ_M: Mod(IMP) → $2^{\mathrm{Mod(EXP)}}$ [ONS 91]. If M1=(IMP1,EXP1) and M2=(IMP2,EXP2) and f: IMP1 → EXP2 is a specification morphism then we define the composition M1 $\bullet_f$ M2 as (IMP2,EXP2'), where EXP2' is defined by the following pushout diagram:

$$\begin{tikzcd}
 & \mathrm{IMP1} \arrow[r] \arrow[d, "f"'] \arrow[dr, phantom, "\mathrm{PO}"] & \mathrm{EXP1} \arrow[d] \\
\mathrm{IMP2} \arrow[r] & \mathrm{EXP2} \arrow[r] & \mathrm{EXP2'}
\end{tikzcd}$$

[†] Depto. Leng. y Sist. Informáticos. Universidad del País Vasco. San Sebastian, Spain

[*] Depto. Leng. y Sist. Informáticos. Universidad Politécnica de Cataluña. Barcelona, Spain

At the model level, the constructor associated to M1 $\bullet_f$ M2 is defined by $\kappa \cdot \kappa_{M2}$, where κ is the extension of κ_{M1} with respect to the above pushout diagram.

A software system S is assumed to be a triple (SP, MSPEC, MPROG), where SP is a requirements specification, including the signature of the whole system and the *global properties* that must be satisfied. MSPEC and MPROG are, respectively, (finite) sets of specification and program modules whose signatures are included in the signature of SP. In general, we assume that systems may be *unfinished*. For instance, the signature of SP may not necessarily be the "union" of all the signatures of the modules in MSPEC, i.e. some components of the system may be not specified yet.. Also, every module in MPROG is the *translation* of a module in MSPEC (its implementation in the given programming language) but there may be some modules in MSPEC whose translation is not in MPROG (i.e. modules not yet implemented).

In order to define the semantics of a system S = (SP, MSPEC, MPROG) we consider the modules as "constraints" that must be satisfied by the given models: an SP-model A "satisfies" or "include" a module M = (IMP, EXP), $A \models M$ iff $A|_{EXP} = \kappa_M(A|_{IMP})$, i.e. if the EXP-part of A is the result of *applying* κ_M to the corresponding IMP-part. If S is finished in the above sense, then its meaning can also be defined in terms of the composition of all the modules in MSPEC. Both semantics can be proved compatible (see [OSC 89] for a special case). In the latter case, the semantics can also be defined equivalently in terms of the composition of all the modules in MPROG.

We consider three basic operations for system development: adding a new specification module to the given system; adding a new program module "translating" a specification module in the system, and, finally, specifying a *simulation implementation* [ONS 91]. In the latter operation, we assume that given two specification modules M1 = (IMP1, EXP1) and M2 = (IMP1, EXP2) in the system, such that M2 can not be directly translated into the programming language, if we find out that a new specification module M3 is an implementation of M2 using M1 (with respect to some suitable notion of behavioural equivalence [Rei 81, ONS 91] in the given institutions), i.e.

$$\forall A \in \mathrm{Mod}(\mathrm{IMP1}) \quad \kappa_{M3 \bullet M1}(A) =_{Beh} \kappa_{M2}(A)$$

then we can substitute M2 by M3 in the system. For instance, if M2 is a module specifying sets, M1 is a module specifying strings and M3 is a module enriching strings with set operations such that the result of forgetting the string operations in the module M3 • M1 is behaviourally equivalent to M2 then we can substitute the set specification by the two modules M1 and M3.

Unfortunately, it may be shown that the latter operation, in general, does not preserve the consistency of a system, as the following counter-example shows. Let SP1, SP2 and SP3 be the following specifications:

```
SP1 = NAT + sorts  s            SP2 = sorts  nat, s        SP2 = SP2 +  opns  g: s → nat
            opns   a,b: s             opns   a: s                       eqns  g(a) = f(a)
            eqns   a = b
                   f(x) = 0
```

where NAT denotes the specification of the natural numbers. Now let S be the system formed by the modules M1 = (Ø, NAT), M2 = (NAT, SP1) and M3 = (SP2, SP3), then it can be easily shown that the algebra A defined $A_{nat} = N$, $A_s = \{a\}$, $a_A = b_A = a$ and $f_A(a) = 0$, $g_A(a) = 0$ satisfies the three modules (assuming that the constructor associated to a module coincides with the associated free construction). Now, let SP4 be the following specification

```
SP1' = NAT +  sorts  s
              opns   a,b: s
                     f: s → nat
              eqns   f(x) = 0
```

and let M2' be the module (NAT,SP1'). Now, assuming that nat is an observable sort and s is non-observable, M2' is a correct simulation implementation of M2. However, there does not exist any algebra satisfying M1, M2' and M3.

This seems to be against the principle of modularity. However, let us assume that the given programming language is stable in the following sense (related notions of stability [Sch 87, ST 89] are essentially equivalent):

Given a program module M = (IMP, EXP) if A is behaviourally equivalent to A', with $A, A' \in Mod(SP)$, for a specification (or program) SP including IMP, and if the amalgamated sums $A+_{A0}A1$ and $A'+_{A0'}A1'$ can be built (i.e. the language allows the pushout associated to these amalgamations), where $A0 = A|_{IMP}$, $A0' = A|_{IMP}$, $A1 = \kappa_M(A0)$ and $A1' = \kappa_M(A0')$ then $A+_{A0}A1$ and $A'+_{A0'}A1'$ are behaviourally equivalent.

Then, the problems shown in the above counter-example are not really important. In particular, if S' is a finished system, obtained after a series of correct development steps from a system S, then the meaning of the composition of all the modules in MPROG is a *realization* of S, even if some "intermediate" systems were inconsistent.

Theorem

Let S = (SP, MSPEC, MPROG) be a consistent system and let S' = (SP', MSPEC', MPROG'), with $SP \subseteq SP'$ be a finished system obtained after applying a sequence of translation and implementation steps over S, then $Sem(S') = \{A \in Mod(SP) / \ \forall M \in MPROG' \ \ A \models M\} \neq \emptyset$.and, in addition, $\forall A \in Sem(S')\ \exists B \in Sem(S)$ such that

$$A|_{SP} \equiv_{Beh} B$$

ACKNOWLEDGEMENTS

This work has been partially supported by ESPRIT Basic Research Working Groups CCL (Ref. 6026) and COMPASS (Ref. 6112).

References

[BH 93] M. Bidoit, R. Hennicker A general framework for modular implementations of modular system specifications, in Prc. TAPSOFT 93 (Orsay, France), Springer LNCS, 1993.

[DGS 91] R. Diaconescu, J. Goguen, P. Stefaneas: Logical support for modularisation, PRG Oxford University, August 1991.

[EBCO 91] H. Ehrig, M. Baldamus, F. Cornelius, F. Orejas: Theory of algebraic module specifications including behavioural semantics and constraints, Proc. AMAST 91, to appear in Springer.

[EM 85] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specifications 1*, Springer 1985.

[EM 89] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specifications 2*, Springer 1989.

[GB 84] J.A. Goguen, R.M. Burstall: Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. Springer LNCS 164, 221-256 (1984).

[ONS 91] F. Orejas, M. Navarro, A. Sánchez: Implementations and behavioural equivalence: a survey. Invited Lecture. 8th Workshop on Specification of Abstract Data Types (Dourdan, 1991). To appear in Springer LNCS.

[OSC 89] F. Orejas, V. Sacristan, S. Clérici: Development of algebraic specifications with constraints. *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology*. Springer LNCS 393, 102-123 (1989).

[Rei 81] H. Reichel: Behavioural equivalence - a unifying concept for initial and final specification methods. *Proc. 3rd. Hungarian Comp. Sci. Conference*, 27-39 (1981).

[Sch 87] O. Schoett: Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis; Report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[ST 88] D.T. Sannella, A. Tarlecki: Toward formal development of programs from algebraic specifications: implementations revisited. Extended abstract in : *Proc. Joint Conf. on Theory and Practice of Software Development*, Pisa, Springer LNCS 249, 96-110 (1987); full version in Acta Informatica 25, 233-281 (1988).

[ST 89] D.T. Sannella, A. Tarlecki: Toward formal development of ML programs: foundations and methodology. LFCS Report Series Department of Computer Science, University of Edinburgh ECS-LFCS-89-71(1989).

INTERACTION BETWEEN ALGEBRAIC SPECIFICATION GRAMMARS AND MODULAR SYSTEM DESIGN

Hartmut Ehrig[1] and Francesco Parisi-Presicce[2]

[1] Fachbereich Informatik
Technische Universität Berlin
D-100 Berlin 10 Germany
[2] Dipartimento di Matematica Pura ed Applicata
Universitá degli Studi L'Aquila
67100 L'Aquila Italy

For the last 20 years, abstract data types have been (usefully) described using algebraic specifications, within different frameworks including equational *institutions*, and with diverse semantics, from initial to loose to stratified to behavioral. An extension of the original formulation which allows to isolate a *variable* part (thus generalizing parametrized specifications) and to *hide* details of the specification from the outside has been defined in [9, 1, 5]. A module specification in such a framework consists of four parts : an export interface EXP specifying what the module specification *produces*, an import interface IMP describing what the module specification *consumes* or needs, a body part BOD describing how the module specification uses the imported items to construct the exported ones, and a parameter part PAR which can be *actualized* by different actual specifications and which is left unchanged by the semantics of the module specification, which is a functorial transformation from the category of models of IMP to the category of models of EXP. The different parts are related by four specification morphisms (usually inclusions) as in the following diagram

$$\begin{tikzcd}
\mathrm{PAR} \arrow[r, "e"] \arrow[d, "i"'] & \mathrm{EXP} \arrow[d, "v"] \\
\mathrm{IMP} \arrow[r, "s"] & \mathrm{BOD}
\end{tikzcd}$$

The items *not* in the image of v are to be considered hidden and thus not visible from other modules. Each module is seen as a self-contained unit which can be developped independently and interconnected with other modules. Three basic interconnection mechanisms have been defined to construct complex systems : a *union* $MOD1 +_{MOD0} MOD2$ where each part is the union of the corresponding ones in MOD1 and MOD2, identifying the MOD0 part; an *actualization* $act_h(PS, MOD)$, where a parametrized specification PS is substituted via the specification morphism h for PAR in each component of MOD; and a *composition* $MOD1 \circ_h MOD2$, where the import IMP1 is matched via h with the export EXP2. Each interconnection can be viewed as an operation on module specifications which preserves correctness and which produces a module specification whose semantics can be expressed in terms of those of the operands.

Given a library LIB of module specifications, an important problem is to determiue whether there is a way to interconnect a subset of *LIB* so that the import and export interfaces of the overall system are some given specifications BASE and GOAL. This problem has been tackled in [11, 12, 13] by considering the visible part of MOD, i.e., the specifications PAR, IMP and EXP, as a production $p : IMP \leftarrow PAR \rightarrow EXP$ similar to those of the algebraic theory of graph grammars [2]. A direct derivation $p : SPEC \Rightarrow SPEC'$ with such a production p is a double pushout diagram

$$\begin{tikzcd}
\mathrm{IMP} \arrow[d] & \mathrm{PAR} \arrow[l] \arrow[r] \arrow[d] & \mathrm{EXP} \arrow[d] \\
\mathrm{SPEC} & \mathrm{CON} \arrow[l] \arrow[r] & \mathrm{SPEC'}
\end{tikzcd}$$

where CON is the context specification unchanged by the *transformation*. Denote as usual by $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$.

It has been shown in [12, 13] that, given a library LIB of module specifications represented by their interfaces $PRO = \{p_i : IMP_i \leftarrow PAR_i \rightarrow EXP_i, i \in I\}$, then $BASE \Rightarrow^* GOAL$ using PRO *if and only if* there exists an interconnection, using only actualization and composition with identity, of (some of) the module specifications of LIB such that BASE and GOAL are the overall import and export interfaces, respectively. This result can be seen as a way to construct a prototype of a system which, given a (built-in) realization of BASE, provides a realization of GOAL (which can be used to test the adequacy of the specification) since there is a systematic way of translating a derivation sequence $BASE \Rightarrow^* GOAL$ into the appropriate combination of the interconnections.

The *initial* item BASE and the set of productions PRO define a grammar, an **algebraic specification grammar** [6] which generates a language $\mathcal{L}$(BASE,PRO) whose membership problem corresponds to the realizability of GOAL.

This solution is not satisfactory for two reasons : the first one is that not every interconnection can be obtained from a derivation sequence (in particular, general compositions with non-identity matching morphism $IMP1 \rightarrow EXP2$) ; the second one is that an occurrence morphism $IMP \rightarrow SPEC$ does not guarantee the applicability of the production [11], while it should be possible to use only part (namely $h(IMP)$) of SPEC as input of MOD. This suggests the notion of *restricting derivation sequences* $SP0 \gg SPn$ with

$SP0 \leftarrow SP1 \Rightarrow SP1' \leftarrow SP2 \Rightarrow \ldots \leftarrow SPn$

where, having generated $SP0$, we can generate $SP1'$ provided that there exist $SP1$ and a specification morphism $SP1 \rightarrow SP0$ such that $SP1 \Rightarrow SP1'$. We then have

Theorem 1. *$BASE \gg GOAL$ via the productions PRO*
if and only if
there exists an interconnection with general composition and actualization using LIB with overall interfaces BASE and GOAL.

It is direct to show that if $SP \Rightarrow^* SP'$, then $SP \gg SP'$ but not necessarily viceversa and it has been shown [14] that the immediate extension of restricting derivation sequences to graphs is more expressive than the single pushout approach to graph transformations [10].

The approach can be extended is a straigthforward manner to High Level Replacement Systems [3]. Such systems have been defined to generalize, in an axiomatic way to arbitrary categories, the Parallelism Theorem, the Concurrency Theorem and other similar results typical of the algebraic theory of graph grammars [2]. By selecting a distinguished subset M of morphisms to be used in the productions, High Level Replacement Systems can be classified (at least) as HLR0, HLR0.5, HLR0.5*, HLR1 and HLR1* depending on which set of generic properties on the underlying category they satisfy and which are sufficient to guarantee properties such as local confluency of independent derivations (Church-Rosser property) or replacing a sequence $p_1 \circ p_2$ of independent derivations with one step using their disjoint union $p_1 + p_2$ (Parallelism Theorem).

Recently [4], results on canonical derivation sequences for graph grammars have been extended to HLR systems. It has been shown that canonical derivations exist for HLR0.5 systems and are unique for HLR1* systems. A canonical derivation is a derivation which does not contain two steps $p_1 : G0 \Rightarrow G1$ and $p + p_2 : G1 \Rightarrow G3$ where $p_1 : G0 \Rightarrow G1$ and $p : G1 \Rightarrow G2$ are sequentially independent. For non canonical derivations in which such a situation occurs, the application of p could be shifted *earlier* to obtain the equivalent derivation sequence $p + p_1 : G0 \Rightarrow G2$, $p_2 : G2 \Rightarrow G3$ which increases the leftmost parallelism. Equivalence of derivations is defined as the reflexive, symmetric and transitive closure of the shift relation.

Of the three possible ways of defining the category of algebraic specifications considered [6, 4] only the one which allows to distinguish, through labels, equations between terms is the one which guarantees unique canonical derivations, while if the specification morphisms $f : SPEC1 \rightarrow SPEC2$ *are such that* $f\#(E1)$ *is either* derivable or contained in $E2$, then every derivation has an equivalent canonical one, which is not necessarily unique.

Canonical derivation sequences can be used to check the equivalence of modular systems. There are several ways of defining equivalence between modular systems, among which:

- $S1 equiv_1 S2$ if the *flattened* versions obtained by applying as operations the interconnections have isomorphic interfaces
- $S1 equiv_2 S2$ if, in addition, the corresponding semantical functors are naturally isomorphic
- $S1 equiv_3 S2$ if the flattened versions are isomorphic

Having the possibility of defining a unique canonical equivalent structuring of a system allows the testing for the equivalence of two arbitrary systems by comparing their canonical forms. By using $equiv_1$ as our notion of equivalence and by limiting the systems to using only disjoint union, actualisation and identity composition and the most restrictive form of specification morphisms (called SPEC3 in [7]) we have

Theorem 2. *Every modular system has a unique canonical equivalent one*

We expect to be able to extend this result to equivalence according to $equiv_3$ and to systems built using general composition.

References

1. E.K.Blum, H.Ehrig, F.Parisi-Presicce: Algebraic Specifications of Modules and their Basic Interconnections. J. Comput. Syst. Sci. 34 (1987) 293-339
2. H.Ehrig: Introduction to the Algebraic Theory of Graph Grammars. First International Workshop on Graph Grammars, Springer Lecture Notes in Computer Science 73 (1979) 1-69
3. H.Ehrig, A.Habel,H.-J.Kreowski,F.Parisi-Presicce: Parallelism and Concurrency in High Level Replacement Systems. Math. Struct.in Comp. Science 1 (1991) 361-406
4. H.Ehrig,H.-J.Kreowski, G.Taentzer: Canonical Derivations for High-level Replacement Systems. Techn.Report 6/92, Univ. Bremen, FB Mathematik und Informatik, Dec 1992
5. H.Ehrig, B.Mahr: Fundamentals of Algebraic Specification 2.Module Specifications and Constraints. EATCS Monograph on Theoretical Computer Science,vol 21,Springer Verlag 1990
6. H.Ehrig, F.Parisi-Presicce: Algebraic Specification Grammars: A Junction between Module Specifications and Graph Grammars. Proc. 4th Int. Workshop on Graph Grammars, Springer Lecture Notes in Computer Science 532 (1991) 292-310
7. H.Ehrig, F.Parisi-Presicce: High Level Replacement Systems for Equational Algebraic Specifications. Proc. 3rd Int. Conf. on Algebraic and Logic Programming, Springer Lecture Notes in Computer Science 632 (1992) 3-20.
8. H.Ehrig, M.Pfender, H.J.Schneider: Graph Grammars : an algebraic approach. Proc. IEEE Conf. SWAT 73, Iowa City (1973) 167-180
9. H.Ehrig H.Weber: Algebraic Specification of Modules. in "Formal Models in Programming" (E.J.Neuhold and G.Chronist, eds.), North-Holland (1985) 231-258
10. M.Lowe: Extended Algebraic Graph Transformation. Doctoral Dissertation, Technische Universitat Berlin Feb 1991, 180 pages
11. F.Parisi-Presicce: Modular System Design applying Graph Grammar Techniques. Proc. ICALP 89, Springer Lecture Notes in Computer Science 372 (1989) 621-636
12. F.Parisi-Presicce: A Rule Based Approach to Modular System Design. Proc. 12th Internat. Conf. Soft. Engin. (1990) 202-211
13. F.Parisi-Presicce: Foundations of Rule-Based Design of Modular Systems. Theoret. Comp. Sci. 83 (1991) 131-155
14. F.Parisi-Presicce: Single vs. Double Pushout Derivations of Graphs. Proc. 18th Int. Workshop on Graph Theoretic Concepts in Comp. Sci. Springer Lecture Notes in Computer Science 657 (1993) 248-262

This article was processed using the LaTeX macro package with LLNCS style

Specification of Hybrid Systems in CRP

R.K. Shyamasundar
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay 400 005, India
e-mail: shyam@tifrvax.bitnet

Abstract

Concurrent languages can be broadly categorized into:

1. *Asynchronous*: A program is a set of loosely coupled independent execution units or processes, each process evolving at its own pace. Interprocess communication is done by mechanisms such as message passing. Communication as a whole is *asynchronous* in the sense that an arbitrary amount of time can pass between the desire of communication and its actual completion. This class includes languages such as Ada, Occam, CSP etc.

2. *Synchronous*: Here, programs are thought of as reacting instantaneously to its inputs by producing the required outputs. Statements evolve in a tightly coupled input-driven way and communication is done by instantaneously broadcasting, the receiver receiving a message exactly at the time it is sent. Languages such as Esterel [BeGo 92], Lustre, Signal, Statecharts belong to this category.

Recently, we have proposed [BeRaSh 93] a new programming paradigm called Communicating Reactive Processes (CRP) [BeRaSh 93] that unifies the capabilities of asynchronous and synchronous concurrent programming languages with a view to specify complex reactive systems which usually have both synchronous and asynchronous features. A CRP consists of a network of Esterel programs where each node can be considered to be reactively driving a part of a complex network that is handled globally by the network. The central idea of establishing asynchronous communication between nodes lies in extending the asynchronous interaction into a communication primitive. The usual **send** and **receive** asynchronous operations are represented by particular tasks that handle the communication. A spectrum of message passing types such as non-blocking **send** for full asynchrony, or CSP-like **send** and **receive** primitives etc. are possible[1] through CRP.

In this paper, we show that

- CRP can model asynchronous systems operating in dense real-time domains, and

[1] In the full paper, we will discuss the above classification in the context of cooperation-synchronism and communication-synchronism and the expressivity of CRP with reference to such a classification.

- CRP can model "continuous" computations and thus, provides a convenient formalism for specifying hybrid systems.

A broad structure of the paper is given in the following.

1 Hybrid Systems

Hybrid systems are systems that combine discrete and continuous computations. To represent continuous computations, hybrid system model contains activities that modify their variables continuously over intervals of positive duration, in addition to the familiar transitions that change the values of variables instantaneously, representing the discrete components. It should be obvious that many systems that interact with a physical environment such as a digital module controlling a process or a manufacturing plant, a digital-analog guidance of transport systems, a control of a robot etc., can benefit from the more detailed modeling proposed by the comprehensive framework of the hybrid model. Various abstract models for systems for handling real-time and "continuous" computations have been proposed recently in [KePn92, MaPn92, NSY92]. There have been several definitions of hybrid systems. One of the definitions corresponds to specifying behaviour sequences explicitly denoting the absence/presence of signals at the timed transitions. Our notion of hybrid systems corresponds to the one defined in [KePn92] based on *hybrid traces*. In this paper, we adapt the CRP [BeRaSh 93] formalism for the specification of hybrid systems and show that it provides a convenient vehicle for specifying hybrid systems. In addition to the implementation of CRP on top of Esterel, the tools and environment of Esterel can be effectively used for the development and verification of CRP programs.

2 Behavioural Specification of Clocked CRP Programs

We start with the addition of the `tick` signal in the behaviour specification of programs as in the case of semantics for the hardware implementation of Esterel [Be 92]. Due to the limited space, we will highlight informally the main features. A brief look at the execution history of a CRP program provides some understanding of the main aspects of CRP and hybrid computations.

A *history* (Esterel or CRP node) is a sequence of events $E_1, E_2, \cdots, E_n, \cdots$; for convenience, we denote E_i by $I_i.O_i$ where I_i and O_i denote the input and the output events in the ith instant respectively. In a *clocked CRP* program every instant consists of the input signal `tick`.

A history is said to be *CRP valid* if it satisfies the following properties:

1. The history satisfies all the declared exclusion relations.

2. $\forall i$, $\texttt{tick} \in I_i$.
   Every input instant contains the special signal `tick`.

   - When programming digital circuits it will naturally denote clock ticks (which corresponds to integer domain of time).

- For dense-domains for time,
  - we consider the signal tick with values. For instance, the signal tick(v) in I_n could indicate an elapse of $v \in \mathcal{R}$ units of time from the last occurrence of tick.
  - further, the sequence of tick's with values forms a progressive sequence; that is, it does not form a Zeno sequence.

3. Asynchronous signals satisfy:
   - An event is received only after requested.
   - The start- and receive- of an asynchronous request cannot happen in the same instant.

4. $\forall i$, $|I_i| + |O_i| < \infty$. That is, it satisfies the property of *finite variability*, namely, the state changes only finitely often throughout any finite interval of time. That is, between any two consecutive input instants containing tick there can be only a finite sequence of events.

We adapt the clocked CRP behavioral specification for hybrid system specification and establish that CRP provides a convenient description for hybrid systems permitting the use of verification tools for Esterel/CRP by:

1. Restricting the behaviour specification for *progressive systems*, i.e., systems that do not admit Zeno sequences.
2. Relating the asynchronous signals with a finite set of continuous activities.
3. Relating the behaviour specification of clocked CRP to the two broad types of computations based on *hybrid traces* (cf. [MaPn 92]):
   (a) Sampling computations having the signature $N \mapsto \Sigma \times R^+$ where each natural number, j, is mapped to a pair consisting of a state s_j and a real-time stamp t_j.
   (b) Super-dense computations[2] having the signature $R^+ \times N \mapsto \Sigma$; that is, it maps each pair $< t, i >$, where $t \in R^+$ and $i \in N$, to a state $s \in \Sigma$ and the step numbers correspond to the transitions that are taken at time instant t.
4. Relating causally correct clocked CRP programs to *timed graphs* and *finitely satisfiable TCTL* formulas.

3 Illustrative Examples

We illustrate specification of hybrid systems through clocked CRP by the following examples:

[2] The advantages/disadvantages of super-dense computation semantics and sampling computation semantics will be discussed in the full paper.

1. We describe the Cat and Mouse problem (cf. [MaPn 92]) and show, how the CRP formalism provides a convenient description including the priority for the Mouse (or the cat) when they reach the target/destination simultaneously.

2. Next, we consider the specification of a controller for controlling the flight path of a communication satellite. Due to various uncertainties at the various stages (due to energy and other motor characteristics) of the launch, it is not possible to pre-program the flight-path of the rocket so as to result in the desired end-conditions within the specified tolerances. Thus, there is a need to determine the flight path from *instant* to *instant* to keep the flight path within the specified tolerance limits. Hence, the control needs to be asynchronous (where events can happen arbitrarily close to each other). We show that clocked CRP provides a convenient formalism for specifying such hybrid systems.

The paper concludes with a discussion of the relative comparison of the formalisms for hybrid specifications such as variants of Statecharts and other formalisms, and also the use of Esterel tools in the development of CRP programs.

Acknowledgment

It is a pleasure to thank Professor Amir Pnueli whose lectures on hybrid systems at TIFR clarified various subtle aspects of hybrid system specification.

References

[Be 92] G. Berry (1992), *A hardware implementation of pure Esterel*, *SADHANA*: Special Issue on Real Time edited by RK Shyamasundar, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences, 17 (1):95-139, 1992.

[BeGo 92] G. Berry and G. Gonthier (1988), *The Esterel synchronous programming language: Design, semantics, Implementation*, Rapport de Recherche 842, INRIA 1988, Science of Computer Programming, Vol. 19, No.2, Nov. 92, pp. 87-152.

[BeRaSh 93] G. Berry, S. Ramesh and R.K. Shyamasundar, *Communicating Reactive Processes*, 20th Acm Symposium on Principles pf Programming Languages, South Carolina, Jan. 1993.

[KePn 92] Y. Kesten and A. Pnueli, *Timed and hybrid Statecharts and their textual presentation*, LNCS 571, pp. 591-619, 1992.

[MaPn 92] Z. Manna and A. Pnueli, *Models for Reactivity*, TR, Stanford, 1992 (an earlier version presented at the 25th Anniversary of INRIA)..

[NSY 92] X. Nicollin, J. Sifakis, and S. Yovine, *From ATP to timed graphs and hybrid systems*, LNCS 600, pp. 549-572, 1992

Real-Time Program Synthesis from Specifications

Aurel Cornell[†], John Knaack[‡], Amitabh Nangia[†], Teodor Rus[‡]
[†] Brigham Young University, Department of Computer Science, Provo, Utah
[‡] The University of Iowa, Department of Computer Science, Iowa City, IA 52242

1 Introduction

Real-time systems are characterized by their umbilical connection to the environment [Wirt77]. They are most often modeled as event driven systems where the occurrence of events dictates a timed response by the system [Dasa85, Jaff91]. Such a behavior is naturally described by a state transition diagram [Best91]. The goal of this paper is to initiate the development of an algebraic methodology for real-time program development that is convenient for the programmer and allows easy proof of the correctness of real-time programs. This methodology is algebraic in nature in the sense that program development is closer to the development of algebraic computations rather than to the development of programs using conventional languages. Following this methodology a real-time program is developed in two steps: first the behavior of a real-time system is specified using a real-time system specification language and then, this behavior is automatically transformed into a semantic driven automaton [Rus91, Knaa92] that implements the real-time program. Consequently, no programming activity in the usual sense is involved. A similar approach for real-time program development is described in [Nico92]. The difference is that our real-time system specification language is a regular language on the alphabet of conditional-actions similar to guarded-commands [Dijk75] while in [Nico92] a language of timed processes [Nico91] is used. In addition, the abstract time used to model the time is different in the two approaches.

2 Real-Time System Specification by Regular Expressions

We consider that any specification language must be provided with a capability for abstraction manipulation that consists of: a mechanism for type definition, a mechanism for object declaration, and a mechanism for application specification. We propose a language capable of specifying the behavior of a real-time application where each specification contains two sections: a declaration section which specifies the types and variables of those types that will be used, and a behavior specification section which specifies the application in terms of the declared variables, as seen below. The declaration section specifies the *state* of the system seen as the interpretation of the collection of names used in the real-time system, $\sigma : Names \rightarrow Values$. Let Σ be the collection of states of a given real time system. The behavior of the system is stepwise specified in terms of two well-understood constructions:

- State transitions, $\tau : \Sigma \rightarrow \Sigma$, expressed by named conditional actions of the form $\tau : \langle Condition \rightarrow Action \rangle$ interpreted as "when Condition holds the Action is performed".
- Composition of state transitions using regular operators. That is, if a and b are state transitions then (1) the concatenation of a and b denoted $a\ b$, (2) the choice of a or b denoted $a|b$, and (3) the repetition of a state transition a known as Kleene star and denoted by a^* are state transitions.

2.1 Declaration Section

The real-time system specification language discussed here allows one to express the behavior of the real-time system as a continuous interaction between the system and its environment. During this interaction the system receives data from the environment that determines the system state transitions; during the state transition the system may receive or send data in the environment affecting it as well as change the state. This behavior is accomplished by considering the environment as a collection of typed communication channels. The basic typed channels are abstract and predefined and are represented by the usual types **boolean, integer, character, string**, etc., whereas the channels specific to the real-time applications are defined in the application and their objects are constructed in terms of the basic type objects. For each channel α and variable x of type α the following operations may be defined: $send(x,\alpha)$ that sends the value of x on the channel α, $receive(x,\alpha)$ that assigns to x the current value on the channel α, and $set(x,t)$ that sets the value of the variable x to a value t of type α, denoted by $x := t$. The constructed types specific to real-time applications considered here are:

Time: In this paper time is an algebraic structure $\mathbf{T} = \langle T, 0, +, \leq\rangle$ as in [Nico92] where

1. $\langle T, 0, +\rangle$ is a commutative monoid such that $\forall t_1, t_2 \in T\ [t_1 + t_2 = t_1]$ implies $t_2 = 0$.
2. $\leq$ is a total order on T such that $\forall t_1, t_2 \in T\ [t_1 \leq t_2] \equiv \exists t_3 \in T\ [t_1 + t_3 = t_2]$.

Practically, time is implemented by a channel on which time is continuously ticking. That is, at any instance a *receive* operation on this channel returns an object of type time that represents the real-time elapsed from the starting of the application until the moment when the *receive* operation has been executed. Note that $send(x, time)$ is not defined. If x is of type *Time* where *Time* is a channel of type time then $set(x,t)$ assign the value t of type time to x and $x := receive(Time)$ sets x to the value $t + \delta$ where δ is the real time elapsed since the last set of x to the time t. The functioning model of a real-time system in this paper is considered as taking place in real time and independently of the time. That is, while state transitions of the real-time system take place as described by the equations specifying the system, the time continuously ticks independently. The time becomes visible only when a variable x of type *Time* is checked by an $x := get(Time)$ operation.

Text: An object of type **text** is a string.

Analog: This type describes analog devices that are controlled by the real-time system, or that are used to sense the real world. A get operation on an analog channel returns an analog value that represents the current measured quantity as sensed by the device, such as temperature, density, speed, etc. A *send* operation on an analog channel may activate a device such as a stepping motor or audible tone generator.

Digital: This type describes digital devices used to sense the environment, or to be controlled by the environment. A digital input channel may be a switch whose status can be read and that is operated by a human, and a digital output channel may be a control which starts a fan, turns on a heating element, etc.

The variable in terms of which system behavior is specified are of the types of channels defining the system. The variable declarations in a real-time system are analogous to variable specifications in a conventional language, i.e. variables are defined over several types (some of them unique to real-time systems). Since all these types are predefined, we need merely to declare a variable of one of these types by specifying a name representing an instantiation of the type.

2.2 Behavior Specification Section

The behavior of the real-time application is represented by a system of conditional equations [Knaa92, Rus92] specifying a semantic driven automaton, denoted by *SDA*. Since the primary behavior of an SDA depends on the content of the data it receives from its environment, the SDA must check conditions on these data in order to know when to move from one state of the real-time application machine to the next and also what actions to perform as it changes state. That is, a state transition is a description of the conditions that allow a move from one state to another. The description consists of a test of the real-world conditions in the application's environment, and of the actions to perform while changing state.

A *condition* is a predicate which tests properties of a message from a channel. For example, let *Temperature* be a channel and *Temp* be a variable of type *Temperature*. Then if the SDA must perform some action if the temperature exceeds 350 degrees, it must perform $Test = get(Temperature)$ and then if $Temp > 350$ the SDA will change state performing some specified action, such as shutting off a heating element, as it does so. Conditional expressions [Knaa92] consist of either single conditions or conditions connected by logical operators *and*, *or* and *not*. Thus, information from multiple channels can be tested in the same conditional expression.

The primitive actions performed by an SDA are:

1. send data x to the channel α, $send(x, \alpha)$.
2. receive data x from a channel α, $x := receive(\alpha)$.
3. Do nothing, denoted by *idle*. This action never terminates. The only way to get out of it is through a preempt operation [Kest92].
4. Skip action denoted by *skip*. This action does nothing and terminates in a single execution.
5. Wait for a condition C to be satisfied denoted by *wait(C)*. This action terminates only when C becomes true.
6. Assignment action denoted by $x := E$ where x is a variable, E is an expression (possibly including a function call) and the type of x is the same as the type of E.

A transition has two parts:

1. A conditional expression which is a predicate over the language of conditions on variables of the types defining the real-time system. An empty condition is interpreted as always true.
2. A list of actions to perform when the condition is satisfied. This can be a (possibly empty) list of *send, receive*, and assignment statements which can modify the value of variables.

A transition in this language has the form *id* : ⟨*condition* → *actionlist*⟩ where *id* is a transition name, *condition* is a conditional expression as defined above, and *actionlist* is a list of actions which actually control the channels making up the real-time system. In this manner each transition defined is represented by a name. The collection of transition names used in a real-time specification is called the alphabet of transitions.

The specification of a real-time system consists of a set of conditional equations over the alphabet of transitions. The equations are of the form *id* = *regular expression* where *id* is either a transition name, one of the names "Start", "Stop", or "Error", denoting the start state, terminate state, and calling an error manager, respectively, or the left-hand side of a previously defined equation. The *regular expression* utilizes the usual regular operations of concatenation, choice and Kleene star over the set of transition names to specify a finite-state machine.

A complete example follows. It is provided by the set of equations that specifies the behavior of an oven [Corn92]. The environment is specified by the following table:

Channels	Type	Names	Values
Time	time	Timer, Update	Integers denoting seconds
Temperature	analog	Temp	Integers denoting centigrades
Commands	digital	Cmd, Heat, Cool	Integers denoting booleans
Predefined	integer	Hyst, SetP,	Integer numbers
Text	Message	M1, M2	Text of messages

The transition equations describing the oven behavior are:

$$
\begin{aligned}
T_1 &: \langle \rightarrow Heat := 0; Timer := 0; Hyst := 1; SetP := 100; Update := 1\rangle \\
T_2 &: \langle Timer = 0 \textit{ and } Cmd = 1 \rightarrow \textit{ skip}\rangle \\
T_3 &: \langle Temp \leq SetP - Hyst \rightarrow \textit{send}(M1, Text); Heat := 1; Cool := 0\rangle \\
T_4 &: \langle Heat = 1 \rightarrow Cool := 0; Timer := Update; Heat := 1; M1 := \textit{receive}(Text)\rangle \\
T_5 &: \langle Heat = 0 \rightarrow Cool := 0; Heat := 1; \textit{send}(M2, Text)\rangle \\
T_6 &: \langle Temp \geq SetP + Hyst \rightarrow Cool := 1; Heat := 0; \textit{send}(M1, Text); Timer := Update\rangle \\
T_7 &: \langle Cool = 1 \rightarrow Cool := 0; Heat := 0; Timer := Update; M1 := \textit{receive}(Text)\rangle \\
T_8 &: \langle Cool = 0 \rightarrow Cool := 1; Heat := 0; \textit{send}(M2, Text)\rangle \\
T_9 &: \langle Timer = 0 \textit{ and } Cmd = 0 \rightarrow Heat := 0; Cool := 0; Timer := 0; \textit{send}(M1, Text)\rangle \\
T_{10} &= (T_3(T_4(T_2|T_9))|(T_5(T_5|T_4))) \\
T_{11} &= (T_6(T_7T_2)|(T_8(T_7|T_9))) \\
Start &= (T_1(T_2(T_{10}|T_{11}|T_9))|(T_9(T_2|T_9)))^*
\end{aligned}
$$

3 Expressive Power of Semantic-Driven Automata

A semantic driven automaton is controlled by the properties of the tokens it recognizes instead of being controlled by their syntax. The semantic driven automaton that recognizes transition equations specifying the real-time system is described by a two-level transition table [Knaa92] and is equivalent to the program controlling the real-time application. Since time is a type of a real-time system, variables of type time can be defined and initialized in the systems specification part. These variables can be set, their values can be tested and used in various conditions defining the transitions of the real-time application. Therefore, a semantic driven automaton that uses time-conditions in its transitions is a timed-automaton [Nico92]. However, any kind of real-time device and condition can be easily integrated in the real-time specification language defined in this paper. Therefore, the semantic driven automata provide a unifying mechanism for real-time program synthesis from specification.

In order to show the expressive power of semantic driven automata we will sketch here the proof that if a computation can be expressed using a conventional language then that computation can be expressed by conditional equations specifying a semantic driven automaton. For that we will consider the statement as the unit of computation specified by conventional languages and will show that any construct expressing control-flow on statements can be expressed by regular expressions using conditional expressions.

Let S_1 and S_2 be statement labels and E be a boolean expression. Then we have:

1. The *concatenation* of S_1; S_2 is a regular expression $S_1\ S_2$.

2. The *branching* statement $if\ E\ thenS_1\ elseS_2$ can be expressed by the regular expression $S' : \langle E \rightarrow S_1 \rangle, S'' : \langle \neg E \rightarrow S_2 \rangle, S = S'\ S''$.

3. The *while loop* $while\ E\ do\ S_1$ can be expressed by the regular expression $S : \langle E \rightarrow S_1 \rangle$, $S' = S^*$.

We used here only regular operators to express conditional equations due to their well understood semantics and well-known methodology of mapping regular expressions into programs. However, the approach we use to implement semantic driven automata allows us to use other operators than the regular ones and therefore we can easily generalize introducing equations of the form $S = S_1 || S_2$ where $||$ denotes the parallel composition of S_1 and S_2 thus obtaining a mechanism for parallel program synthesis from specifications.

The major advantages of this methodology for program synthesis from specifications are:

1. It allows stepwise program development in terms of simple actions, well-understood by programmers, and their automatic composition through the mechanism of transforming regular expressions in programs.

2. It allows formal proof of the program correctness by first proving the correctness of the simple actions making up the program and by automatically preserving this correctness by the translator mapping regular expressions in programs.

3. The automatic mapping of regular expressions in efficient programs is feasible and well-understood.

4. It unifies the methodology of program synthesis from specification, and open new field of research.

References

[Best91] Bestavros, A., "Specification and Verification of Real-Time Embedded Systems using Time Constrained Reactive Automata", *Proceedings IEEE 12th Real-Time Systems Symposium*, Dec 4-6 1991, San Antonio, Texas, 244-253.

[Corn92] Cornell, A., "Oven", *Research Report* 7, CS Department, Brigham Young University, 1992.

[Dasa85] Dasarathy, B., "Timing Constraints of Real-Time Systems", *IEEE Transactions on Software Engineering*, Volume 11, Number 1, 1985, 80-86.

[Dijk75] Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", *Communications of the ACM*, Volume 18, Number 8, 1975, 453-457.

[Jaff91] Jaffe, M. S. et al, "Software Requirements Analysis for Real-Time Process-Control Systems", *IEEE Transactions on Software Engineering*, Volume 17, Number 3, 1991, 241-257.

[Kest92] Kesten, Y., Pnueli, A., "Timed and Hybrid Statecharts and their Textual Representations", *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 571, 1992, 591-619.

[Knaa92] Knaack, J. and T. Rus, "TwoLev: A Two Level Scanner", *Proceedings of AMAST'91*, Workshops in Computing Series, Springer Verlag 1992, 264-276.

[Nico91] Nicollin, X., Sifakis, J., "An Overview and Synthesis on Timed Process Algebra", *Proceedings Third Workshop on Computer-Aided Verification*, Alborg, Denmark, July 1991, 1-21.

[Nico92] Nicollin, X., Sifakis, J., Yovine, S., "Compiling Real-Time Specifications into Extended Automata", *IEEE Transactions on Software Engineering*, Volume 18, Number 9, 1992, 794-804.

[Rus91] Rus, T., "Algebraic Construction of Compilers", *Theoretical Computer Science*, Volume 90, 1991, 271-308.

[Rus92] Rus, T., "Computation Specification by Semantic Driven Automata" Unpublished Paper, The University of Iowa, Department of Computer Science, Iowa City, IA 52242, 1992.

[Shaw92] Shaw, A. C., "Communicating Real-Time State Machines", *IEEE Transactions on Software Engineering*, Volume 18, Number 9, 1992, 805-816.

[Sifa92] Sifakis, J. et al, "Compiling Real-Time Specifications into Extended Automata", *IEEE Transactions on Software Engineering*, Volume 18, Number 9, 1992, 794-804.

[Wirt77] Wirth, N., "Toward a Discipline of Real-time Programming", *Communications of the ACM*, Volume 20, Number 8, 1977, 577-583.

On the coverage of partial validations

Ed Brinksma
Tele-Informatics and Open Systems Group,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
brinksma@cs.utwente.nl

Extended Abstract

1 Introduction

It is widely recognized that a completely rigourous treatment of the correctness of designs of realistic information-processing systems is beyond the scope of the formal methods currently at our disposal. While for some aspects of this predicament improvements can be expected through the development and use of more powerful formalisms, theories, and supporting software tools, there are structural problems related to managing the combinatorial complexity of correctness proofs for large systems. The investment done to prove correctness should also be measured against the remaining possibility that errors are introduced in the ultimate realization phase of the design, where formal methods may no longer be applicable (e.g. silicon compilation). As a result in practice mostly methods that deal with *approximate* correctness criteria are used, such as testing and the verification of particular properties. This should also be seen against the background that complete correctness of systems in not required in most cases: in reality one tries to make the occurrence of important errors sufficiently unlikely.

In recent years there has been also a growing theoretical interest in the question of partial validation, which has led to much new work on topics like *model checking* and *testing theory*, e.g. [9, 5, 3, 2]. This has given rise to new algorithms for the validation of given properties and for the generation of tests, whereas the related question concerning the *coverage* of partial validation methods, i.e. how much the validation of a particular property contributes to the overall correctness the design, has received considerably less attention. Such measures are needed to guide the selection of properties that should be validated, and can be used to quantify the quality of a validation procedure, and, indirectly, of the implementations that succeed in passing them. Unfortunately, it is not straightforward how to obtain such measures.

In the literature it has been proposed to use the probability of the occurrence of an error as a guiding principle for partial validations, viz. by ignoring improbable errors (see e.g. [7]). This would seem to suggest that the coverage of such a partial validation method could be calculated as a sum of the probabilities of independent errors that are exposed by the method. This approach has the drawback that often it may be the absence of less frequent errors that determines the quality of a design. Moreover, the appreciation of the role of a particular error may depend more on the application of a system than on its specified abstract functionality. The occurrence of the same software bug in a computer game and in the operating system of a nuclear plant could be of a radically different importance, and this should preferably be reflected in a coverage measure.

In an earlier paper we have therefore proposed that coverage should be based on so-called *valuations* that assign weights to error classes corresponding to their gravity [4]. Probability distributions over error classes being special instances of valuations, this yielded a generalization of the probability induced notion of coverage. This approach has the drawback, however, that it was not clear how, in view of their subjective nature, such valuations could be obtained or approximated for given applications. For the probability induced notion of coverage there exists at least the wealth of statistical methodology to estimate the distributions involved.

In the full version of this paper we refine our idea of valuations. Measurements of the probability of error occurrences in implementations can be used to improve our estimates of valuations, while still maintaining a possibility to account for the difference between the probability and the gravity of an error. We follow a measure-theoretic approach in which an exogenous *cost* function (quantifying the effect of certain properties in an implementation) is integrated over a measure that is induced by the probability of error occurrences. In this way, in fact, we do not only obtain a notion of coverage, but a general way of assigning measures to specification theories in the context of a given class of implementation structures.

2 Main formalizations

The correctness of an implementation I with respect to a specification S is usually formalized by means of an *implementation* or *refinement* relation R such that I is a correct implementation of S iff $\langle I, S\rangle \in R$. We will in fact assume that this relation R can be formalized in terms of the satisfaction of a logical theory, viz. $R = \{\langle I, S\rangle \mid I \models Th(S)\}$, where $Th(S)$ is the theory in some logical language $\mathcal{L}$ specified by S and $\models$ denotes a satisfaction relation. Many implementation relations can in fact be characterized in this way, including those using constructive specification formalisms (see e.g. [8]).

As indicated above we view the design process as a stochastic experiment that produces an implementation $\underline{I}$ on the basis of a given specification S. In order to model this correctly we need to define a Borel space in which $\underline{I}$ takes its value (see e.g. [1]). Let $\mathcal{I}$ be the set of all potential implementations of S, and $\Phi_{\mathcal{L}}$ the set of formulae in $\mathcal{L}$ then we are particularly interested in the sets

$$V_{\Phi} =_{df} \{I \in \mathcal{I} \mid I \models \Phi\} \quad \text{for } \Phi \subseteq \Phi_{\mathcal{L}} \tag{1}$$

We say that $\mathcal{I}$ has the *Borel* property w.r.t. $\mathcal{L}$ iff $\mathcal{V} =_{df} \{V_{\Phi} \mid \Phi \subseteq \Phi_{\mathcal{L}}\}$ is a Borel set for $\mathcal{I}$, i.e. (i) $\emptyset \in \mathcal{V}$, and ($ii$) $\mathcal{V}$ is closed under arbitrary unions and complementation (w.r.t. $\mathcal{I}$). Requirement (i) is easily fulfilled, viz. if $\mathcal{L}$ is sufficiently rich to allow for *inconsistent* theories Φ, as that implies $V_{\Phi} = \emptyset$. The closure property w.r.t. complementation is more involved as for each $\Phi \subseteq \Phi_{\mathcal{L}}$ there need not exist a $\Phi' \subseteq \Phi_{\mathcal{L}}$ such that $V_{\Phi'} = \overline{V_{\Phi}} = \{I \in \mathcal{I} \mid I \not\models \Phi\}$. As the latter set could be characterized by the disjunction over the negations of all $\varphi \in \Phi$, one solution would be to work with languages that have either explicit generalized disjunctions, such as e.g. $\mathcal{L}_{\omega_1\omega}$ [6], or implicit ones, e.g. in the form of fixpoint constructions [8]. Another option is to restrict the class of implementations $\mathcal{I}$. In practice, for example, one can often restrict the attention to a *finite* set $\mathcal{I}$ where each $I \in \mathcal{I}$ is completely characterized by a *finite* theory $\Phi_I \subseteq \Phi_{\mathcal{L}}$. In that case ordinary negation and disjunction suffice to warrant the closure properties.

Assuming that $\mathcal{I}$ has the Borel property w.r.t. $\mathcal{L}$ we can now introduce for each specification S a measure P_S over $\mathcal{I}$, viz. by putting

$$P_S(V_\Phi) \ =_{df} \ Pr\{I \in V_\Phi\} \tag{2}$$

i.e. assigning V_Φ the probability that the implementation satisfies Φ.

As we have observed above we wish to modify this measure by also taking into account the *cost* of errors. We assume therefore there exists a function $k : \mathcal{P}(\Phi_\mathcal{L}) \rightarrow \mathbf{R}_{\geq 0}$ that determines the cost $k(\Phi)$ of satisfying the properties of Φ. This function has to satisfy the intuitive property that cost increases with logical strength, i.e. $\Phi \models_\mathcal{I} \Psi$ implies that $k(\Phi) \geq k(\Psi)$, where $\Phi \models_\mathcal{I} \Psi$ means that for all $I \in \mathcal{I} \models \Phi$ implies $I \models \Psi$. If we put $Th(I) =_{df} \{\varphi \in \Phi_\mathcal{L} \mid I \models \varphi\}$ then we can overload k to include a function of type $\mathcal{I} \rightarrow \mathbf{R}_{\geq 0}$ by putting $k(I) =_{df} k(Th(I))$. It can be shown quite easily that this function is integrable w.r.t. each measure P_S. This result allows us to define the *valuation* measure μ_S on $\mathcal{V}$ as the measure-theoretic integral

$$\mu_S(V) \ =_{df} \ \int_{\overline{V}} k(I)\, dP_S \quad \text{for all } V \in \mathcal{V} \tag{3}$$

Note that in order to calculate $\mu_S(V)$ we integrate the cost of its complement $\overline{V}$. This can be understood by realizing that once we have established, by (partial) validation, that $I \models \Phi$, or equivalently that $I \in V_\Phi$, it follows that $I \notin \overline{V_\Phi}$ so that the cost related to implementations in $\overline{V_\Phi}$ has been avoided. This seems a natural way to measure the value of having established Φ. Another way of looking at it is that μ_S must increase with logical strength, as k does: if $\Phi \models_\mathcal{I} \Psi$ then Φ contains more information than Ψ, and should consequently have a higher valuation. This follows as $\Phi \models_\mathcal{I} \Psi$ implies $V_\Phi \subseteq V_\Psi$ implies $\overline{V_\Psi} \subseteq \overline{V_\Phi}$ implies $\mu_S(V_\Psi) \leq \mu_S(V_\Phi)$.

Because of the non-continuous nature of $\mathcal{I}$ the integral in (3) will in practice be evaluated as a, possibly infinite, summation. Nevertheless, equation (3) gives us the most compact representation of the definition of the measure in full generality.

Having established the measure μ_S for given specifications S it is now straightforward to produce the definition of the coverage of a partial validation w.r.t. S as a normalization of μ_S. Let $\Phi \subseteq Th(S)$ then a procedure for establishing that $I \models \Phi$ has a *coverage* α, with $0 \leq \alpha \leq 1$, iff

$$\mu_S(V_\Phi) \ \geq \ \alpha . \mu_S(V_{Th(S)}) \tag{4}$$

We also say that an implementation I is α-correct, or, alternatively, has an margin of error of $1 - \alpha$, iff there exists a $\Phi \subseteq Th(S)$ with $I \in V_\Phi$ for which equation (4) holds. Note that 1-correctness does not necessarily coincide with total correctness in the classical sense, as errors with measure 0 are ignored if the measure that is used admits their existence.

It should be noted that the above definition of coverage applies even in the pathological case where $\mu_S(V_{Th(S)}) = 0$, by (4) trivially yielding coverage 1 for any Φ. In the normal case, i.e. when $\mu_S(V_{Th(S)}) \neq 0$, the normalisation can be applied directly to the definition of the measure itself by putting

$$\mu^*_S(V) \ =_{df} \ \mu_S(V)/\mu_S(V_{Th(S)}) \tag{5}$$

In this way μ^*_S has become insensitive to the absolute value of applications of the cost function k, taking only its proportional variation into account. Inequality (4) then simplifies to $\mu^*_S(V_\Phi) \ \geq \ \alpha$.

In the full paper we give an elaborated example of the application of our theory, which is an extension of the probabilistic example in [4]. Of course, an important point in the application of this theory is how to obtain reliable estimates of $Pr\{\underline{I} \in V_\bullet\}$. The solution here probably lies in measuring error distributions that result from the application of individual design steps that are applied sufficiently often to obtain statistical significance, as opposed to complete design procedures for entire systems, which are often specific for the particular system that is designed. By calculating the cumulative effect of the applied design steps still a reasonable error distribution estimate could be obtained. Not surprisingly reliable coverage measures are thus tied to the application of well-understood design methods. Of course, the theory can be used also to give coverage assessments under the hypothesis of given error distributions. By making such assumptions explicit more precision is given to the coverage claims that are made.

References

[1] H. Bauer, *Probability Theory and Elements of Measure Theory*, Holt, Rinehart, and Winston.

[2] G. Bernot, Testing against formal specifications: a theoretical view. In: S. Abramsky and T.S.E. Maibaum (eds.), *TAPSOFT'91*, Volume 2, 99–119. LNCS 494, Springer-Verlag, 1991.

[3] E. Brinksma, A Theory for the derivation of tests. In: S.Aggarwal and K. Sabnani (eds.), *Protocol Specification, Testing, and Verification VIII*, 63–74, North-Holland, 1988.

[4] E. Brinksma, J. Tretmans, and L. Verhaard, A framework for test selection. In: B. Jonsson, J. Parrow, and B. Pehrson (eds.), *Protocol Specification, Testing, and Verification XI*, 233–248, North-Holland, 1991.

[5] P. Godefroid and P. Wolper, Using Partial orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: K.G. Larsen and A. Skou (eds.), *Computer Aided Verification '91*, 332–342. LNCS 575, Springer-Verlag, 1992.

[6] H.J. Keisler, *Model Theory for Infinitary Logic*, North-Holland.

[7] N.F. Maxemchuk and K. Sabnani, Probabilistic Verification of Communication Protocols. In: H. Rudin and C. West (eds.), *Protocol Specification, Testing, and Verification VII*, North-Holland, 1987.

[8] C. Stirling, Modal and Temporal Logics for Processes, *LFCS Report Series*, ECS-LFCS-92-221, Dept. of Computer Science, University of Edinburgh, 1992.

[9] A. Valmari, Error detection by reduced reachability graph generation. In: *Proc. 10th International Conference on Application and Theory of Petri Nets*, volume 2, 1–22, Bonn, 1989.

Verifying communication protocols via testing-projection

Khalil DRIRA, Pierre AZEMA
e-mail: {khalil,azema}@laas.laas.fr
LAAS-CNRS, 7 avenue du Colonel Roche F-31077 Toulouse Cedex

1 Introduction

Protocol projection is an efficient approach for the analysis of communication protocols. It consists of deriving from an initial automaton the minimal one, while preserving specific equivalence relations. According to the OSI model, the basic architecture consists of two protocol entities communicating via an underlying service. The global service, the protocol provides, corresponds to the behaviour as observed from both Service Access Points only. Using LOTOS notations, this service can be described by the following expression where SAP_i are the service access points, and Γ_i designates the interaction point that synchronizes entity i and the communication medium:

$$\begin{aligned} \mathit{service}[SAP_1, SAP_2] = \ & \text{hide } \Gamma_1, \Gamma_2 \text{ in} \\ & ((\mathit{entity}[SAP_1, \Gamma_1] \ ||| \ \mathit{entity}[SAP_2, \Gamma_2]) \\ & \qquad |[\Gamma_1, \Gamma_2]| \mathit{medium}[\Gamma_1, \Gamma_2]) \end{aligned}$$

Compiling such expression produces a Labeled Transition System (LTS) which describes the service. This LTS is generally so complex that it is very difficult for the designer to decide whether this service is the expected one for the specified protocol.

Verifying a protocol specification can be carried out by using an equivalence relation. The specification is correct when the provided service is equivalent to the expected one. This equivalence-based verification approach is well known (see for example [4]).

In practice, it is not always possible to have a reference model (the so-called expected service). This is due to the difficulty to describe this service in a monolithic style where composition operator is not used. Projection is then a convenient alternative to protocol verification. It consists of furnishing a reduced model of the protocol service while preserving some properties: the more the properties are strong the more the reduced model is complex.

Such a reduced model can be obtained using equivalence induced by the weak bisimulation (the observational equivalence) [6, 8]. Projection of LTS according to this equivalence produces a minimal LTS in the sense that it contains no bisimilar distinct states. The size of the so-reduced system can be such that it is still not possible to analyze it. Currently, this problem is solved by substituting trace equivalence (known also as language equivalence in the automata theory) for observational equivalence. The projection supplies a deterministic LTS (with only observable events) which is minimised w.r.t. the bisimulation equivalence (see Fig. 1). Unfortunately, the size reduction is accompanied by a loss of preserved properties: the only preserved properties are those concerning event ordering. That is the reduced system accepts the same strings (sequences of events) as the initial system.

We propose here a tradeoff between the complexity of the reduced system and the properties this system preserves. We propose to define a new projection relying on another equivalence. This equivalence is known as testing equivalence ($\underline{te}$) in Brinksma's testing theory for LOTOS [1] and is a simplification of Hoare's failure equivalence used for CSP [5]. This equivalence is less discriminating than observational equivalence but more discriminating than trace equivalence. It preserves the traces and the failures of a system, that is properties dealing with the possibilities of deadlock with the system environment.

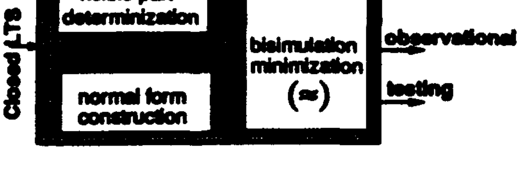


Figure 1: *Trace, observational and testing projections. The closed LTS is obtained by considering* $\stackrel{a}{\Rightarrow} = \xrightarrow{\tau^* a \tau^*}$

Unlike bisimulation-based equivalences it is incorrect to minimize the system by identifying testing equivalent states. This is always the case for non-bisimulation-based equivalences. We propose here to solve this difficulty for the testing equivalence. For this purpose a transformation (designated as *normal form* for testing equivalence) of LTS is defined. This transformation simplifies an LTS and preserves the testing equivalence. This transformation is defined using recursive algebraic definitions. This makes it support rigorous and simple proofs of correctness.

This paper is composed of this introduction and four other sections. The next section recalls standard definitions related to LTSs and testing equivalence. The testing projection is then introduced. Before conclusions, we compare the three projections on a small simple example.

2 Basic definitions

Labeled Transition Systems (shortly LTS) are the basic structure commonly used to represent dynamic behavior of communicating systems.

A Labeled Transition System can be viewed as a set of processes (S) executing actions in Σ. The behavior of a process $s \in S$ is specified by the set of actions it can perform. The behavior following an action is specified by the set of transitions Δ.

A finite Labeled Transition System (LTS) is a quadruple: $S = (S, \Sigma, \Delta, s_0)$ where:

- S is a finite set of states, and s_0, $s_0 \in S$, is the initial state of S.
- Σ is a finite set of visible actions, or labels
- $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$: the transitions set, $\tau \notin \Sigma$ is called internal or invisible · ction. An element $(x, \mu, y) \in \Delta$ is denoted: $x \xrightarrow{\mu} y$

Another transition relation, $\{\overset{\mu}{\Rightarrow}\}_{\mu \in \Sigma \cup \{\varepsilon\}}$ is defined in a standard way by:

- $s \overset{\varepsilon}{\Rightarrow} s'$: $s = s'$ or $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s'$: this means that internal moves of a system cannot be distinguished.
- $s \overset{a}{\Rightarrow} s'$: $s \overset{\varepsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\varepsilon}{\Rightarrow} s'$: this means that observable moves are not distinguished by internal moves that encapsulate them.

The output of a state s denotes the set of visible actions that can be performed by the system at the state s. Formally $out(s) = \{a \in \Sigma \mid s \overset{a}{\Rightarrow}\}$.

This relation is extended to sequences (i.e. words or strings over Σ: $\sigma \in \Sigma^*$) by:

•if σ is the sequence $a_1 \cdots a_n$ write $s \overset{\sigma}{\Rightarrow} s'$ when $s \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \cdots \overset{a_{n-1}}{\Rightarrow} s_{n-1} \overset{a_n}{\Rightarrow} s'$

The empty sequence is denoted ε. As in the case of a state output, "traces of a state" refer to the set of all sequences of (visible) actions, $\sigma \in \Sigma^*$, that can be performed from this state: $Tr(s) = \{\sigma \in \Sigma^* \mid s \overset{\sigma}{\Rightarrow}\}$. By convention, the traces of LTS are those of its initial state: $Tr(S) = Tr(s_0)$.

Using LTS, we recall now the formal definition of conformance introduced in the testing theory of LOTOS [1]

Definition 1 (testing equivalence [1]) *Two LTSs $P_i = (S_i, \Sigma, \Delta_i, P_i)$ $i = 1, 2$ are said to be testing equivalent (noted P_1 $\underline{te}$ P_2) when*
(1) $Tr(P_1) = Tr(P_2)$
(2) $\forall \sigma \in \Sigma^*, \forall A \subseteq \Sigma$
$\exists P_1', P_1 \overset{\sigma}{\Rightarrow} P_1'$ and $\forall a \in A \neg(\exists P_1'', P_1' \overset{a}{\Rightarrow} P_1'')$ iff
$\exists P_2', P_2 \overset{\sigma}{\Rightarrow} P_2'$ and $\forall a \in A \neg(\exists P_2'', P_2' \overset{a}{\Rightarrow} P_2'')$

□

3 The testing projection

The testing projection of LTS S is an $\approx$-minimized normal form of this system: $(nf(S))_{\approx}$, where $\approx$ is the bisimulation equivalence.

The resulting system verifies the following expected properties:

- A projection of an LTS is testing equivalent to this LTS: $(nf(S))_{\approx}$ $\underline{te}$ S
- Two testing-equivalent LTSs have the same projection. In other words testing equivalence is an isomorphism over the subset of LTSs which are $\approx$-minimal normal forms.

Definition 2 (LTSs in normal form) *An LTS, $S = (S, L, \Delta, s_0)$, is said to be in normal form for the testing-equivalence if*
(i) *its initial state s_0 verifies the following equation:*

$$s = \Big(\sum_{X \in R(s)} \tau; \sum_{a \in out(s) \setminus X} a; f_a(s)\Big) \; [] \sum_{b \in \bigcap_{X \in R(s)} X} b; f_b(s) \qquad \textbf{(P1)}$$

Where $R(s)$ is a (non-empty) set of subsets of $out(s)$ which verifies the following minimality (w.r.t. cardinality) property:

$$\forall X, Y \in R(s) : (Y \subseteq X) \Rightarrow (X = Y). \qquad \textbf{(P2)}$$

(ii) *The states, $f_a(s)$ et $f_b(s)$, specified in equation* **P1** *verify also* (i) *et* (ii). □

The Lotos operators ";" and "[]" designate respectively action prefix and choice. The Lotos expression $\sum_{i \in I = \{1, \cdots, n\}} P_i$ denotes the expression $P_1[]P_2[] \cdots []P_n$. Semantic of these operators is defined by the following rules:

$$\forall a \in \Sigma \cup \{\tau\} \quad \text{(i)}\ \frac{}{a; P \xrightarrow{a} P} \qquad \text{(ii)}\ \frac{k \in I, P_k \xrightarrow{a_k} P_k'}{\sum_{i \in I} P_i \xrightarrow{a_k} P_k'}$$

The following proposition shows that testing equivalence and observational equivalence (i.e. weak bisimulation equivalence) are identical over LTSs in normal form.

Proposition 3 *If S_1 et S_2 are in normal form (def. 2) then* $(S_1 \text{ te } S_2) \Leftrightarrow (S_1 \approx S_2)$ ■

Definition 4 (Refusal Graph) *A refusal graph, denoted RG, is a bilabeled graph represented by a 5-tuple $(G, \Sigma, \Delta, g_0, Ref)$ where:*
- *(G, Σ, Δ, g_0) is a deterministic LTS. That is which verifies : $\forall g \in G$, $\forall a \in \Sigma$; $\exists$ at the most one $g' \in G$ such that $(g, a, g') \in \Delta$. This successor can then be noted $f_a(g)$ which means that the set of transitions is described using a family of functions $\{f_a : G \longrightarrow G\}_{a \in \Sigma}$.*
- *$Ref : G \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$ is an application which defines for each state, the sets of actions that may be refused after the sequence leading to this state.* □

To avoid redundancy, refusal sets must be minimal w.r.t. set inclusion: $\forall g \in G, \forall X, Y \in Ref(g) : (Y \subseteq X) \Rightarrow (X = Y)$.

And to avoid describing imaginary systems, the following hypotheses is imposed on the refusal graph structure: $\forall X \in Ref(g)$, $X \subseteq out(g)$. Only refused parts of the output set are considered.

Let $\mathcal{S}$ be the transition system (S, Σ, Δ, s_0) and the two following applications, whose domain is the set of subsets $\mathcal{P}(S)$,

$$\delta_a(P) = \bigcup_{p \in P} \delta_a(p) \quad \text{and} \quad out(P) = \bigcup_{p \in P} out(p)$$

where P is a subset of S, and $\forall s \in S$, $\delta_a(s) = \{s' \in S | s \overset{a}{\Rightarrow} s'\}$.

Definition 5 ("rg" transformation)
The refusal graph $rg(\mathcal{S})$, associated with transition system $\mathcal{S} = (S, \Sigma, \Delta, s_0)$ is defined by the 5-tuple $\mathcal{G} = (G, \Sigma, \Delta', Ref, g_0)$, where

- $g_0 = \delta_\epsilon(s_0) = \{s \mid s_0 \overset{\epsilon}{\Rightarrow} s\}$
- *$\langle G \subseteq \mathcal{P}(S), \Sigma, \Delta' \subseteq G \times \Sigma \times G \rangle$ is the labeled graph $rg(g_0)$, where for all $g \subseteq S$, $rg(g)$ is recursively defined by the following Lotos expression :*

  $$rg(g) = \sum_{a \in out(g)} a; rg(\delta_a(g))$$

- *and for all $g \in G$, $Ref(g) = \{out(g) \setminus out(s), s \in g\} \setminus \{X \in Ref(g), \exists Y \in Ref(g) : (X \subseteq Y \text{ and } X \neq Y)\}$*

□

Definition 6 ("lts" transformation) *From a refusal graph g_0, an LTS $lts(g_0)$ may be derived according to the following recursive definition :*

$$lts(g) = \sum_{X \in Ref(g)} \tau; \sum_{a \in out(g) \setminus X} a; lts(f_a(g))$$
$$[] \sum_{b \in \bigcap_{X \in Ref(g)} X} b; lts(f_b(g))$$

□

Note that the parameterised bisimulation of [2] can be used to provide a decision procedure for testing equivalence using bisimulation over refusal graphs. Bisimulation over refusal graphs is an interesting question in its own right and will not be further explored in this paper.

Definition 7 ("nf" transformation) *The normal form of an LTS $\mathcal{S}$ is the LTS $nf(\mathcal{S})$ derived from the refusal graph of $\mathcal{S}$, that is $rg(\mathcal{S})$, by using transformation lts. That is : $nf(\mathcal{S}) = lts(rg(\mathcal{S}))$.*
□

Remark: In the case of strongly convergent (i.e. when no loop is created by internal transitions), the "nf" transformation is identical to the transformation described in [7].

Theorem 8 *Every LTS is testing-equivalent to its normal form: $\mathcal{S}$ te $nf(\mathcal{S})$.* ■

The next proposition can be deduced from the proposition 3 and the theorem 8. It provides an alternative (to the Π-bisimulation of [2]) of verification of testing equivalence allowing (weak) bisimulation equivalence over standard LTSs to be used.

Proposition 9 $\mathcal{S}_1$ te $\mathcal{S}_2 \Leftrightarrow nf(\mathcal{S}_1) \approx nf(\mathcal{S}_2)$ ■

Proposition 10 *For every LTS, we have:*
$\mathcal{S} \approx \mathcal{S}_{\approx}$. And
$\mathcal{S}_1 \approx \mathcal{S}_2$ iff $(\mathcal{S}_1)_{\approx} \leftrightarrow (\mathcal{S}_2)_{\approx}$. Where $\leftrightarrow$ denotes the isomorphism over LTSs. ■

Finally, using the fact that $\approx$ is compatible with te (i.e. $\approx \subset$ te) and using the standard results of proposition10, we deduce from proposition 9

Proposition 11 *For every LTS, we have:*
$\mathcal{S}$ te $(nf(\mathcal{S}))_{\approx}$. And
$\mathcal{S}_1$ te $\mathcal{S}_2$ iff $(nf(\mathcal{S}_1))_{\approx} \leftrightarrow (nf(\mathcal{S}_2))_{\approx}$. Where $\leftrightarrow$ denotes the isomorphism over LTSs. ■

4 Example

Figure 2 presents an example of the three former projections. Observational projection does not reduce the initial LTS. This is due to the fact that states s_2 and s_3 are not observationally equivalent because their behaviours are respectively of the form $\tau; (P[]Q)[]\tau; P$ and $\tau; P[]Q$ which are not observationally equivalent.

The system depicted by these LTSs can be viewed as the local service provided by a data transfer connection-oriented protocol which locally uses a

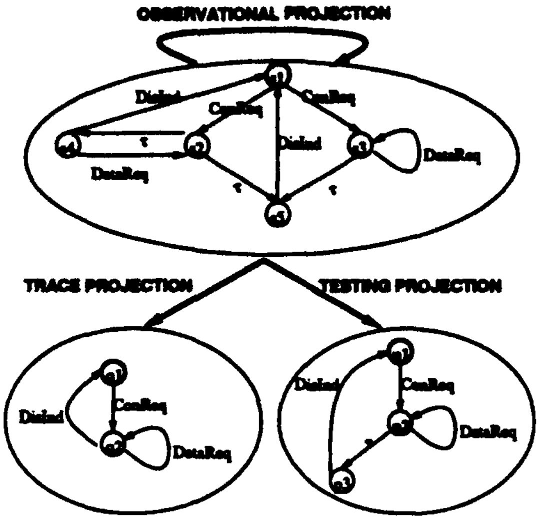


Figure 2: *testing projection provides LTS less reduced than trace projection and more reduced than observation projection*

rendez-vous communication between a protocol entity (i.e., service provider) and its user. The trace projection shows that initially the system can allocate a connection (ConReq), and then transmit data (DataReq) or accepts disconnection (DisReq). The testing projection shows that, after connection, data transmission is not always possible. This is due to the presence of an internal transition ($p_2 \xrightarrow{\tau} p_3$) that system may execute without communicating with its environment compelling the latter to stop data transfer. Abstraction made by this projection consists of ignoring the origin of this internal transition. It can either represent a remote or a local disconnection decision.

5 Conclusion

The underlying idea of the testing-projection can be summarised by the following:

- we characterise a particular family of LTS called LTS in normal form. For this family we prove that (weak) bisimulation equivalence is identical to testing equivalence.
- we provide a transformation of an LTS to a testing equivalent LTS which is in normal form. This transformation relies on an abstract structure (we refer to as Refusal graph [3]) that eliminates redundancy related to information that does not concern trace and deadlock properties.
- ≈-minimization of this normal form preserves testing equivalence and reduce the state space of the LTS.

The minimization part of the testing-projection can be conducted by means of strong bisimulation equivalence. This provides easier minimization and is possible by slightly modifying (the definition of the normal form and) the "lts" transformation.

This technique has been experimented on several communication protocols, namely MMS and OSI-TP. These experiences showed that in the first design steps the so-reduced system is useful for specification error detection and correction.

References

[1] E. Brinksma, G. Scollo, and C. Steenbergen. Lotos Specifications, their implementations and their tests. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification Testing and Verification*, volume VI. Elsevier Science Publishers B.V., North-Holland, 1987.

[2] R. Cleaveland and M. Hennessy. Testing Equivalence as Bisimulation Equivalence. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 11–23, Grenoble-france, June 1989. Springer-Verlag.

[3] K. Drira, P. Azéma, B. Soulas, and A.M. Chemali. Testability of a communicating system through an environment. In *Proc. 4th International Join Conference on the Theroy and Practice of Software Development. TAPSOFT'93 (LNCS 668)*, ORSAY, FRANCE, April 1993.

[4] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A toolbox for the verification of lotos programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, New-York, May 1992. ACM.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[6] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[7] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[8] D. Park. Concurrency and automata on infinite sequences. In *Lecture Notes in Computer Science*, volume 104. Springer-Verlag, Berlin Heidelberg, 1981.

Equivalences of transition systems in an algebraic framework

Pasquale Malacaria[*]

In this paper we study simulation and bisimulation equivalences for transition systems from an algebraic point of view. For the simulation equivalence, the algebras are the free algebras of a monad on the category of transition systems. These algebras are however "concrete" because they are an "algebraic completion" of a system. A more interesting category of algebras seems to be the one we will propose for studying bisimulation; being related to the category of transition systems by a Stone duality it is in some sense canonical. Here by canonical we mean that the algebra associated to a transition system is as close as possible to the structure of the system (roughly speaking it is the space of ultrafilters on the systems). Stone duality makes it possible to establish an equivalence between categories having a very different structure, for example between categories of algebras and categories of topological spaces [5] or between categories of domains and categories of algebras and logics [1, 3]. The Stone duality we present in this paper relates the category of transition systems to a category of algebras underlying a generalised Hennessy-Milner logic [4], that is algebras which contain Lindembaum algebras of this logic. As a test for the validity of abstract reasoning (i.e. algebraic tools) about transition systems, we will prove the equivalence of the notions of subalgebra and bisimulation relation, that is we will prove that two systems are in bisimulation if and only if they have an isomorphic subalgebra. It will follow then that the minimal subalgebra of the algebra of a system T corresponds by duality to the smallest transition system (w.r.t. number of states) which is in bisimulation with T.

1 Categories of action algebras and transition systems

A complete atomic Boolean algebra (CBA for short) is a Boolean algebra A in which the g.l.b. and l.u.b. operations are defined for all subsets of A and such that there exists a subset $\mathrm{At}(A) \subseteq A$ such that $A \simeq \wp(\mathrm{At}(A))$ (the power set of $\mathrm{At}(A)$).

Let **CBA** denote the category whose objects are complete atomic Boolean algebras and whose arrows are the structure preserving maps. Note that if $\phi : A \rightarrow A'$ is an arrow in **CBA** , then there exists a unique set theoretical map

$$\phi^* : \mathrm{At}(A') \rightarrow \mathrm{At}(A)$$

such that (under the isomorphism $A \simeq \wp(\mathrm{At}(A))$) we have $(\phi^*)^{-1} = \phi$. The map ϕ^* is the *underlying map for ϕ*.

Let A be a CBA and X a set; a *linear action* of X on A is given by a map $\alpha : X \times A \rightarrow A$ (we write $x.v$ instead of $\alpha(x, v)$) such that:

- $x.0 = 0$,

[*] LIENS, Ecole Normale Superieure, Paris, France and Imperial College, London, UK. e-mailpasquale@dmi.ens.fr

- $x.\bigvee V = \bigvee_{v \leq V}(x.v)$.

The *category of actions of X over complete atomic boolean algebras* (category denote as $\mathcal{AL}$) has as objects pairs (A, α) (lets call such a pair an *action algebra*) where A is a CBA and α is a linear action of X over A. An arrows $\phi : (A, \alpha) \rightarrow (A', \alpha')$ is **CBA** morphisms between A and A' which satisfy the inequality:

$$x.\phi(v) \leq \phi(x.v)$$

A transition system is a pair $T = (S, T)$ (we use the same letter T to indicate the set of transitions and the transition system) where S is the set of *states* and $T \subseteq S \times X \times S$ is the set o' *transitions* whose elements we denote as $s \xrightarrow{x} s'$. A transition system map f from (S, T) to (S', T') is a set theoretic map $f : S \rightarrow S'$ such that

$$s \xrightarrow{x} s' \in T \Rightarrow f(s) \xrightarrow{x} f(s') \in T'$$

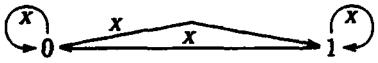
Let $\mathcal{TS}$ denote the *category of transition systems over a set of action X*.
The categories $\mathcal{TS}$ and $\mathcal{AL}$ are related by two contravariant functors $\mathbf{Ts} : \mathcal{AL} \rightarrow \mathcal{TS}$ and $\mathbf{Ac} : \mathcal{TS} \rightarrow \mathcal{AL}$.

- The functor **Ts** is defined as follows:
  $\mathbf{Ts}(A, \alpha) = (\mathrm{At}(A), T_A)$ where $a_1 \xrightarrow{x} a_2 \in T_A$ iff $a_1 \leq x.a_2$
  $\mathbf{Ts}(\phi) = \phi^*$ (the underlying map before defined)
- The functor **Ac** is defined in the following way:
  $\mathbf{Ac}(T) = (\wp(S), \alpha)$ where $\alpha(x, v) = \{s_1 \in S | \exists s_2 \in v \text{ such that } s_1 \xrightarrow{x} s_2\}$
  $\mathbf{Ac}(f) = f^{-1}$

Proposition 1 *The categories $\mathcal{TS}$ and $\mathcal{AL}$ are duals (i.e. $\mathcal{TS} \simeq \mathcal{AL}^{\mathrm{Op}}$)*

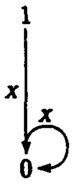
Indeed the duality between $\mathcal{TS}$ and $\mathcal{AL}$ is a Stone duality : Roughly speaking this means that there exists an action algebra $\Omega_{\mathcal{AL}}$ and a transition system $\Omega_{\mathcal{TS}}$ such that the functor **Ts** is naturally isomorphic to the Hom enriched [1] functor $\mathcal{AL}(-, \Omega_{\mathcal{AL}})$ and that the functor **Ac** is naturally isomorphic to the Hom enriched functor $\mathcal{TS}(-, \Omega_{\mathcal{TS}})$.

For example the transition system $\Omega_{\mathcal{TS}}$ is shown in the following picture:



That is $\Omega_{\mathcal{TS}} = (\{0, 1\}, \{s_1 \xrightarrow{x} s_2 | s_1, s_2 \in \{0, 1\}, x \in X\})$

The action algebra $\Omega_{\mathcal{AL}}$ is pictured as follows:



That is $\Omega_{\mathcal{AL}} = (\{0, 1\}, \alpha)$ where α is defined by $x.a = 0$ for $a \in \{0, 1\}$

[1] By "enriched" we mean that the functor $\mathcal{AL}(-, \Omega_{\mathcal{AL}})$ associates to an action algebra A the set $\mathrm{HOM}_{\mathcal{AL}}(A, \Omega_{\mathcal{AL}})$ equipped with a transition system structure

Proposition 2 • *The functors* **Ac** *and* $\mathcal{TS}(-, \Omega_{\mathcal{TS}})$ *are naturally isomorphic.*

- *The functors* **Ts** *and* $\mathcal{AL}(-, \Omega_{\mathcal{AC}})$ *are naturally isomorphic.*

2 Simulation equivalence and Kleisli category

Given two transition systems $T = (S, T), T' = (S', T')$ a *simulation* between T and T' is a relation $\mathcal{R} \subseteq S \times S'$ such that:

(1): For any $s \in S$ there exists $s' \in S'$ such that $(s, s') \in \mathcal{R}$.

(2): For any $s_1 \xrightarrow{x} s_2 \in T$ if $(s_1, s_1') \in \mathcal{R}$ then there exists $s_2' \in S'$ such that $s'_1 \xrightarrow{x} s'_2 \in T'$ and $(s_2, s_2') \in \mathcal{R}$.

A *bisimulation* between T and T' is a simulation $\mathcal{R}$ between T and T' such that $\mathcal{R}^{-1}$ is a simulation between T' and T.

Let consider the functor **Sm** : $\mathcal{TS} \rightarrow \mathcal{TS}$, defined on objects by $\mathbf{Sm}(S, T) = (\wp^+(S), T^+)$ where :

- $\wp^+(S)$ is the set of non empty subset of states of S
- $V_1 \xrightarrow{x} V_2 \in T^+$ iff for any $s_1 \in V_1$ there exists $s_2 \in V_2$ such that $s_1 \xrightarrow{x} s_2 \in T$.

Sm is defined on arrows by $\mathbf{Sm}(f) = f^+$, f^+ being the extension of $f : (S, T) \rightarrow (S', T')$ to the subsets of S. Intuitively the functor **Sm** maps a transition system T in the space of all possible simulations on T.

The functor **Sm** has a natural structure of monad $(\mathbf{Sm}, \eta, \mu)$ so that we can consider the Kleisli category of **Sm** on $\mathcal{TS}$, noted as $\mathcal{TS}_{\mathbf{Sm}}$. We characterise then simulation equivalence as follows:

Proposition 3 *Let T, T' two transition systems. Then there exists a simulation between T and T' iff there exists an arrow between T and T' in $\mathcal{TS}_{\mathbf{Sm}}$.*

3 Action algebras and Bisimulation

A *subalgebra* A' of an action algebra (A, α) is given by a subset of elements of A which is closed under the operations. By using the isomorphism between A and $\wp(At(A))$, we can consider set theoretic operations on atoms of A; hence we define a subalgebra of (A, α) as a subset A' of elements of A such that: For any $v \in V \subseteq A'$ and for any $x \in X$ the elements $\emptyset, A, \bigcup V, \bigcap V, \neg v, \alpha(x, v)$ are in A'

We can prove then:

Theorem 1 *Two transition systems T, T' are in bisimulation iff* $\mathbf{Ac}(T), \mathbf{Ac}(T')$ *have an isomorphic subalgebra.*

The subalgebras of a given algebra are closed under arbritary intersections; in particular the intersection of all subalgebras of A is a subalgebra which is the smallest (w.r.t. inclusion) subalgebra of A. This minimal subalgebra has a very interesting property:

Theorem 2 *Let T be a transition system and let A_0 be the minimal subalgebras of* $\mathbf{Ac}(T)$. *Then the smallest transition system (w.r.t. number of states) which is in bisimulation with T is the transition system* $\mathbf{Ts}(A_0)$

4 Skeleton of an action algebra

Note that in the case of a CBA the notion of minimal subalgebra is trivial, the latter always being the algebra $\{0, 1\}$. The presence of actions in the category $\mathcal{AL}$ makes this notion not trivial since for any $x \in X$ the element $x.1$ (which in general is not 0) must be in the minimal algebra. Hence we are looking for a set Σ_A, the *skeleton* of the algebra A, that is the smallest subset of A containing 1 and closed under linear actions.

Σ_A is included in the minimal subalgebra of A and has moreover a natural structure of transition system (note that Σ_A is a rooted transitions system, the root being the 1 of the algebra).

We define then a *skeleton* homomorphism between two skeletons Σ, Σ' as a transition systems morphism which preserve the root and investigate the equivalence induced by skeleton isomorphism which we note $\simeq_\Sigma$. This is a rather weak equivalence. Indeed we have:

Proposition 4 *Let T and T' be two transition systems such that for any $s \in S$ there exists a trace-equivalent state $s' \in S'$ and for any $s' \in S'$ there exists a trace equivalent state $s \in S$: Then $T \simeq_\Sigma T'$.*

References

[1] S. Abramsky. *Domain theory in logical form.* Proceedings of the 2nd annual symposium on Logic in Computer Science, 1987.

[2] A.Arnold. *Systemes de transitions finis et semantique des processus comunicants.* Masson, 1992.

[3] T.Ehrhard, P.Malacaria. *Stone duality for stable functions* Proceedings of Category Theory in Computer Science, L.N.C.S. 530.

[4] M.Hennessy, R.Milner. *Algebraic laws for nondeterminism and concurrency* Journal of A C M., vol 32, 1985.

[5] P. Johnstone. *Stone Spaces.* Cambridge University Press 1982.

Semantics frameworks for a class of modular algebraic nets

E. Battiston, V. Crespi, F. De Cindio, G. Mauri

Dipartimento di Scienze dell'Informazione - Università degli Studi di Milano
email: decindio@hermes.unimi.it

Among the various proposals for an 'Algebraic Specification of Concurrency' [AR], OBJSA Nets [BDMa] are a class of algebraic high-level nets which combine Superposed Automata (SA) nets, a modular class of Petri nets, and the algebraic specification language OBJ. OBJSA Nets together with their support environment ONE (OBJSA Net Environment), constitute a specification language for distributed systems which is called OBJSAN as each OBJSAN specification is mapped by ONE into an OBJSA Net [BDMb].

To enhance specification modularity and reusability, an OBJSAN specification is obtained by composing, via transition fusion (i.e., superposition), some OBJSAN (open) components. An OBJSAN component is a couple which consists of a net and an OBJ module. The net part expresses the control of the system to be specified and the OBJ part describes data modification through occurrence of events modelled by net transitions. An OBJSAN component is either *closed,* if all of its transitions are closed, or *open* if it contains at least one open transition, i.e., a transition which is only partially extensionally specified, since couples of its input/output places have to be identified through superposition of the transition itself with other transition(s). Open transitions represent the interface of the component toward other components, and are specified by non executable modules (in OBJ called theories), while closed transitions are specified by executable modules (in OBJ called theories).

With the aim of defining a formal semantics for this class of algebraic high–level Petri nets, two operators have been defined in [BDMR]: Spec(_) and Unf(_). They map an OBJSAN closed component (in the following called OBJSAN system) C respectively to an OBJ module Spec(C) called the *Specification module* (by translation of the net scheme into conditional equations and operators) and to a 1–safe SA labelled pure net Unf(C) (an Elementary Net system) called the *Unfolding net* (by translation of the OBJ specification into net elements).
While Unf(_) well supports concurrency since it produces Elementary Net (EN) systems, whose categorical semantics has been defined in [DKPS], Spec(_) is less satisfactory because of the loss of concurrency due to the OBJ3 sequential semantics. The idea is therefore to turn on the specification language MAUDE.
Let us recall that MAUDE is a specification language syntactically similar to OBJ3 whose operational and denotational semantics were defined by Meseguer in [MESa]. In MAUDE there exist essentially two kinds of modules: functional modules (whose syntax is entirely identical to OBJ3) and system modules. While operational semantics is concurrent rewriting for both of kinds of modules, denotational semantics is different. For the functional modules it is the usual initial algebra associated to the equational specification (so MAUDE has OBJ3 as sublanguage). For the system modules it is a categorical model which describes the system whose local behaviour is specified by the rewriting rules.

More precisely let us consider a case that will be useful in the following. Suppose to have a MAUDE system module M which imports a functional module M'. M codes a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ while M' codes a rewrite theory $\mathcal{R}' = (\Sigma', E', L', R')$ where Σ (resp., Σ') is an equational signature, E (resp., E') is a set of Σ_equations, L (resp., L') is a set of labels, R (resp., R') is a set of conditional rewriting rules of the type

l: $[t]_E \rightarrow [t']_E$ if Cond, with $l \in L$ and $[t] \in T_{\Sigma,E}(X)$ (resp. for R'). The operational semantics of the global specification is given by a categorical model in which objects are the elements of $T_{\Sigma \cup \Sigma', E \cup E'}(X)$ and arrows are all the possible sequents $[t]_{E \cup E'} \rightarrow [t']_{E \cup E'}$ inductively generated by the rewriting logic inference rules starting from R∪R'. In practice this means that we have concurrent rewriting modulo E∪E' on terms $T_{\Sigma \cup \Sigma'}(X)$ by using R∪R' as rewriting rules [Mesa], i.e. concurrent rewriting in both the system module (called supermodule in the following) and the functional module (called submodule).
The denotational semantics is given by a categorical model in which the objects are the elements $T_{\Sigma \cup \Sigma', E \cup E' \cup \mathrm{Unlabel}(R')}(X)$ and arrows are all the possible sequents $[t]_{E \cup E' \cup \mathrm{Unlabel}(R')} \rightarrow [t']_{E \cup E' \cup \mathrm{Unlabel}(R')}$ inductively generated by the rewriting logic inference rules starting from R. So the denotational semantics treats the rewriting rules in the functional module as equations whose semantics is the initial algebra. Then only the rewriting rules in the system supermodule are interpreted as arrows (class of closed arrows) of the categorical model.

According to these considertaions, here we redefine Spec(_) as the operator which maps an OBJSAN system C=(N,A) to a MAUDE system module which imports functional modules: a (conditional) rewriting rule in the system module is associated with each transition $t \in T$, while the functional submodules contains the coded specification of the data structure of C (the information in A).

As we are now able to associate a MAUDE module Spec(C) and an EN system Unf(C) with each OBJSAN system C, to give it a semantics we consider the categorical models developed for MAUDE modules (by Meseguer [Mesa], see above) and for Petri nets (by Meseguer&Montanari [MM]) and we verify the isomorphism between the two semantics. As we shall see, both of the categorical semantics result to be redundant. The reason is that OBJSAN systems introduce, for modelling purposes, constraints on the marking: tokens are couples <a_name;some_data>, where the name represents the token identity which cannot change by transition occurrence and is unique in each elementary subnet of an OBJSAN system. Therefore, the net markings are multisets of tokens without multiplicity (i.e., sets) and the Unf(_) operation maps an OBJSAN system C to a contact-free EN system (while proper multisets at the higher level would require a P/T system at the lower level).

$$\begin{tikzcd}
\underline{\text{MAUDE module:}} & & \\
\text{Spec(C)} \arrow[r, "(2)"] & \text{Conditional rewriting theory Rc} \arrow[r, "(6)"] \arrow[dd, "\text{(3) Instantiation theorem.}"] & \text{Full subcategory of } \tau\text{R obtained constrainig logic rewriting inference rules.} \\
\text{C=(N,A)} \arrow[u, "(1)"] \arrow[d, "(5)"] \arrow[ur, dashed, "\text{Lemma 1}"] \arrow[dr, dashed, "\text{Lemma 2}"'] & & \\
\text{Unf(C)} \arrow[r, "(4)"] & \text{Ground rewriting system Tc} \arrow[r, "(7)"] & \text{Partial commutative monoid on a category.} \\
\underline{\text{EN system}} & &
\end{tikzcd}$$

fig 1

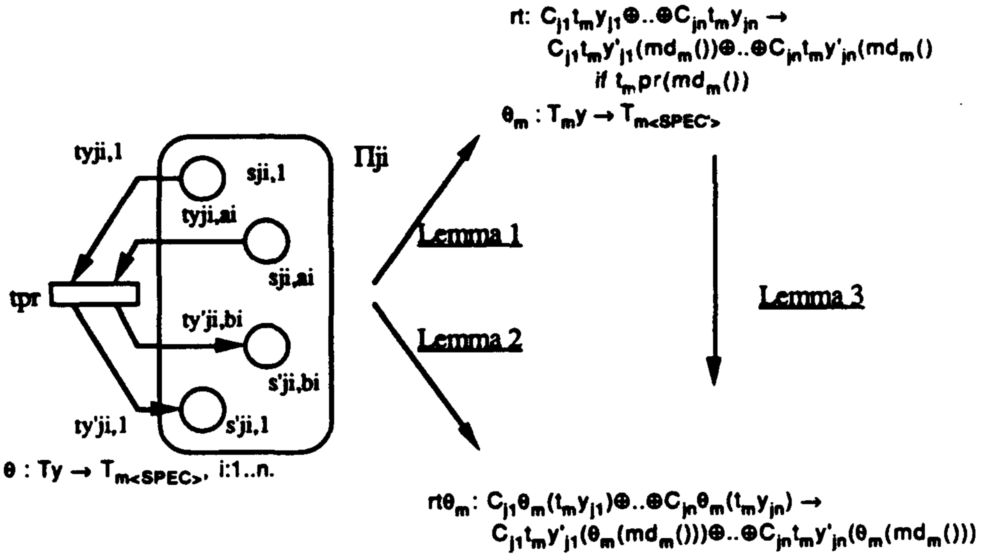
The relationship between Spec(_) and Unf(_) is stated by a theorem that we call 'instantiation theorem' as it proves that by instantiating the rewriting rules of the system part of Spec(C) with ground terms and considering only those rules whose predicates are reduced to true (representing transitions with a chance of occurring), we get the transitions of Unf(C).

More formally, let C=(N,A) be an OBJSAN system and let us derive its MAUDE Specification module Spec(C) and its Unfolding net Unf(C) (arrows 1 and 5 in fig.1).

As we have seen, Spec(C) codes two rewrite theories R=(Σ, E, L, R) and R'=(Σ', E', L', R'), respectively associated with the system module and with the functional submodules. The rewriting theory Rc=(Σ∪Σ', E∪E'∪Unlabel(R'), L, R) (arrow 2) gives the denotational semantics of Spec(C), according to [MESa].

According to the constraction given in [DKPS] which specializes the Meseguer&Montanari work for P/T nets to EN systems, Unf(C) can be translated into a set of ground rewriting rules which we name Tc (arrow 4). For example, a transition t in an EN system is translated in the rewriting rule $s_1 \oplus .. \oplus s_n \rightarrow s'_1 \oplus .. \oplus s'_m$ [commutativity, associativity, identity: λ] where $\bullet t = \{s_1,..,s_n\}$ and $t\bullet = \{s'_1,..,s'_m\}$. Tc gives the denotational semantics of Unf(C), according to [DKPS].

Then, the instantiation theorem (arrow 3) states that by instantiating the open (conditional) rewriting rules in Rc with ground terms and considering only the conditional equations whose predicates are reduced to true we get Tc. In the following we sketch its proof, whose kernel consists of three constructive lemmas related as shown in fig.2.


rt: $C_{j1} t_m y_{j1} \oplus .. \oplus C_{jn} t_m y_{jn} \rightarrow$
$C_{j1} t_m y'_{j1}(md_m()) \oplus .. \oplus C_{jn} t_m y'_{jn}(md_m()$
if $t_m pr(md_m())$
$\theta_m : T_m y \rightarrow T_{m<SPEC>}$
tyji,1
sji,1
Пji
tyji,ai
sji,ai
tpr
ty'ji,bi
s'ji,bi
ty'ji,1
s'ji,1
Lemma 1
Lemma 2
Lemma 3
$\theta : Ty \rightarrow T_{m<SPEC>}$, i:1..n.
$rt\theta_m: C_{j1}\theta_m(t_m y_{j1}) \oplus .. \oplus C_{jn}\theta_m(t_m y_{jn}) \rightarrow$
$C_{j1} t_m y'_{j1}(\theta_m(md_m())) \oplus .. \oplus C_{jn} t_m y'_{jn}(\theta_m(md_m()))$
$t\theta: \theta(ty_{j1,1}) \oplus .. \oplus \theta(ty_{jn,an})) \rightarrow$
$ty'_{j1,1}(\theta(md())) \oplus .. \oplus ty'_{jn,bn}(\theta(md()))$

with $md_m() = mode(t_m y_{j1},...,t_m y_{jn})$, $md() = mode(ty_{j1,1},...,ty_{jn,an})$.

fig 2

Taken a transition t∈T of an OBJSAN system C and a ground substitution θ of the input arc inscriptions (representing an occurrence mode enabling t in a certain marking) we get, via lemma 1, a conditional rewriting rule rt with a corresponding ground substitution θ_m

for var(rt) and, via lemma 2, a ground rewriting rule $t\theta$. Lemma 3 closes the cycle: by instantiating rt with θ_m we get $t\theta$.
The rewriting system Tc is obtained directly by applying lemma 2 to the transitions of C.
The rewrite rules in $Rc=(\Sigma_1, E_1, L_1, R_1)$ are obtained directly by applying lemma 1 to the transitions of C, while the equations E_1 concern the data part of C. As we have said, the MAUDE functional modules specify data, i.e., the tokens carriers, the occurrence predicates $t_m$pr and the arc inscriptions containing variables t_my and operators t_my'(). In fact the carriers of tokens together with the operations defined on them are abstract data types. We instead use system modules to specify control, i.e., local transitions.
The idea is that concurrent term rewriting in system modules captures the concurrency expressed by the control part of the net, while concurrent term rewriting in functional modules performs the parallel computation of the operators ty'. Lemma 3 proves the semantic equivalence between Spec(C) and Unf(C), namely between the concurrency expressed by the system module, captured by Rc, and the concurrency expressed by the Unfolding EN system, captured by Tc. Besides, the lemma proves that the concurrent application of two conditional rewriting rules in Rc r1 and :2 to a marking term s (with *substitutions* θ_{m1} and θ_{m2}) *represents the concurrent occurrence of the two* corresponding low level transitions $r1\theta_1$ and $r2\theta_2$ in Tc in the marking represented by s.

Let us now discuss the redundancy of the two categorical mo('ɔls due to the constraints which characterize OBJSAN systems.
a) The categorical model proposed by Meseguer, when applied to our case (Rc), is redundant because the inductive process generation of the category (by rewriting logic inference rules) would produce arrows without corresponding net computations. We get the correct model constraining the logic rewriting inference rules. What we obtain is a full subcategory of the Meseguer' original model in which objects are associated with admissibles net states only (arrow 6). Such states are denoted by terms not containing two or more identical tokens: this is because OBJSAN system markings do not allow multisets of tokens with multiplicity. From the operational point of view, proofs in this *modified* Meseguer formal system, represent the simultaneous application of several rewrite rules in R_C to a correct marking term, so that the concurrent term rewriting models concurrent transition occurrence. In practice, since we can consider only marking terms s without multiplicity then it is possible to concurrently apply two or more rewriting rules ri of Rc to s if and only if the corresponding matching substitutions θ_{mi} do not share any token (i.e., the occurrence modes are disjoint).
b) The categorical model for P/T nets defined in [MM] is redundant when applied to EN systems, as shown in [DKPS]. The redundancy is eliminated by reducing the parallel sum carrier, leading to a partial commutative monoid on a category (arrow 7) (cf. in [DKPS] the EN category).

By removing the constraints characterizing OBJSAN systems we fall in the more general class of SPEC-inscribed nets [REI] to which Unf(_) and Spec(_) can be extended: in that case the Meseguer and Meseguer&Montanari categorical models would not be redundant. Nevertheless, as a counterpart, SPEC-inscribed nets do not support modularity and therefore they have not a notion of parameterized open component. Indeed, our current effort is extending the approach to OBJSAN open components semantics towards using the categorical frameworks presented here for characterizing concurrent object-oriented languages (cf. [BCDM] and [MESb]).

Ackowledgements

This work has been supported by the ESPRIT Working Groups CALIBAN and ASMICS2 and by the CNR-Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo", sottoprogetto 4, LRC LAMBRUSCO.

References.

[AR] E. Astesiano, G. Reggio, Algebraic Specification of Concurrency. In: Proc 8th WADT, Dourdan (F), LNCS, Springer-Verlag (to appear).

[BCDR] E. Battiston, P. Consolaro, F. De Cindio, L. Rapanotti. POTS, POLS, OBJSA Nets: from object-based to class-based net formalisms. CNR-Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo", Tech. Rep.n.° i/4/59, 1992.

[BDMa] E. Battiston, F. De Cindio, G. Mauri. A class of high level nets having objects as domains. In: G. Rozenberg (ed.), Advances in Petri nets. LNCS 340, Springer-Verlag, 1988.

[BDMb] E. Battiston, F.De Cindio, G.Mauri: Specifying concurrent systems with OBJSA Nets. CNR-Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo", Tech. Rep.n.° i/4/72, 1992.

[BDMR] E.Battiston, F.De Cindio, G.Mauri, L.Rapanotti. Morphisms and Minimal Models for OBJSA Nets. In: Proc.12th Int. Conference on Application and Theory of Petri nets, Gjern (DK) June 1991.

[DKPS] C.Diamantini, S.Kasangian, L.Pomello, C.Simone. Elementary Nets and 2-Categories. CNR-Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo", Tech. Rep. n.° i/4/29, March 1991.

[MESa] J. Meseguer. Conditional rewriting logic, deduction, models and concurrency. In: S.Kaplan and M.Okada, (eds.), Conditional and Typed Rewriting Systems, LNCS 510, Springer-Verlag, 1991.

[MESb] J. Meseguer. Multiparadigm logic programming. In: H. Kirchner and G. Levi (eds), Algebraic and Logic Programming, LNCS 632, Springer-Verlag, 1992.

[MM] J. Meseguer, U. Montanari. Petri nets are monoids. In: Information and Computation, Volume 88, fascicolo 2, 1990.

[REI] W. Reisig. Petri Nets and Algebraic Specifications. In K. Jensen and G. Rozemberg (eds), High-Level Petri Nets. Theory and Applications, Springer-Verlag, 1991.

A Characterization of LOTOS representable Networks of Parallel Processes *

David de Frutos-Escrig
Departamento de Informática y Automática
Facultad de Ciencias Matemáticas - Universidad Complutense
28040 Madrid, Spain

Introduction

We compare in this paper graphic and algebraic representations of parallel networks of processes. More exactly we characterise the class of graphically definable networks that can be represented in LOTOS, that is to say by means of a LOTOS expression which combines the processes in the network by instances of the LOTOS parallel operator.

Unfortunately the obtained characterization is far from trivial, and due to its complexity, even someone could think that it should be not named in such a way. Thus, instead of a characterization we could say that what we have obtained is an efficient algorithm to decide if a given network is representable. But if finally we have decided to insist on the use of the word *characterization*, it is mainly to emphasize the efficiency of the obtained algorithm. Due to the finite nature of the considered problem it is obvious that it is decidable; but at the same time, its combinatorial flavour makes reasonable to expect an exponential complexity, and even to conjecture that the problem was NP-complete. As a matter of fact we thought for a long time that this was the case, once we proved that several, more and more sophisticated, *natural* algorithms to solve the question were not correct.

The work was motivated by our joint work with T. Bolognesi in [3,2], exploring the (partial) associativity properties of the LOTOS parallel operator. First results on the subject, obtained by our colleague, were presented in [1]. Another contribution to the study of the subject is [4], where a simple example proving that no every parallel network is LOTOS representable was presented, but no characterization of the set of representable networks was there presented.

We consider that the work is interesting for several reasons. First it compares two different formal methods for defining concurrent systems: a graphical approach, that we formalize using some basic graph notions; and an algebraic approach, mainly the LOTOS language (or equivalently CSP with its generalized parallel operator.) Characterizing the kind of networks that are LOTOS representable, we show at which extend this kind of graphical representations can be used to specify systems, when we want to use the algebraic framework to analize, or to transform, the obtained specifications. On the other hand, if we focus on the algebraic formalism, we show which are the exact limits of the expressing power of the (LOTOS) parallel operator.

Besides, we consider that the proof of the characterization is rather interesting by itself, showing an application of many different techniques for showing properties of algebraically defined systems. The use of operational semantics, induction, normal forms, reductions of difficult instances of the problem to more simple ones, and some others, are illustrated.

Definitions

In the following we will sketch the main definitions and results to be extended in the full paper.

Definition 1 A general process-gate net GPGN is an undirected bipartite graph (P,G,E), where P is a set of so called process-nodes, G is a set of gate-nodes, and E is a set of arcs $E \subseteq P \times G$. □

Definition 2 A labelled process-gate net LPGN is a triple (GN,GL,AL) where GN = (P,G,E) is a GPGN, $GL : G \rightarrow Gates$ is a labelling function, and $AL : P \rightarrow \mathcal{P}(Gates)$ is a function defining the (maximal) alphabet

*This work has been partially supported by ESPRIT Project 2304: LOTOSPHERE

of the processes to be associated to each process node, such that $\forall (P_i, g) \in E\ GL(g) \in AL(P_i)$, and

$$\forall P_i \in P\ \forall a \in AL(P_i)\ \exists g \in G\ \exists e = (P_i, g) \in E \mid GL(g) = a$$

□

Remark: The last condition in the previous definition is included in order to give a chance to be executed to any appearance of an action in the process labelling each node.

Definition 3 A concrete process-gate net CPGN is an instantiated GPGN, which means a pair (LN,PL) where LN = (GN,GL,AL), with GN = (P,G,E), is a LPGN, and $PL : P \rightarrow Proc$, with $Alphabet(PL(P_i)) \subseteq AL(P_i)$, for each $P_i \in P$. □

This definition is more general that the one given in [3], where GL had to be injective, and each process had to be connected with any gate-node labelled by any of its gates. This restriction was there included (and also in [1]) in order to formalize the so called *maximal cooperation principle*, which obliges to any process including a gate in its alphabet, to cooperate to execute the corresponding action. We have dropped this restriction, what have already been done (although in a different framework), in [2].

Definition 4 (Operational semantics of CPGN's)
Let CPGN = (LN,PL) a concrete process-gate net with LN = (GN,GL,AL) and GN = (P,G,E). For each $g \in G$, if all the processes B_i labelling process-nodes connected with g can execute the action $GL(g)$, evolving into B_i', then CPGN can also execute that action, evolving into CPGN' = (LN,PL'), where PL' is defined as PL, but taking $PL'(P_i) = B_i'$, for each process node P_i connected with g. □

We want to decide if for a given LPGN we can construct an equivalent LOTOS representation LRep(LN), which means a parallel expression combining the process variables in P, by parallel operators $|[S]|$, with $S \subseteq Gates$, such that for any concrete instance of LN, CN = (LN,PL), we have $CN \sim LRep(LN)[PL(P_i)/P_i]$, where by $[B_i/P_i]$ we denote the substitution of all the appearances of the variables P_i by the corresponding processes B_i.

We apply a constructive method to answer the question, so that whenever there exists any LOTOS expression representing the given network, we obtain one of them.

The algorithms to check LOTOS-representability of a network

To solve the problem we have followed a three steps procedure, generalizing at each step the kind of networks that can appear as input.

In the first step we consider the case in which the given network has a single gate. This simple case is studied, just because to solve the general case we have first to solve each of the problems corresponding to the projection of the network over each one of its gates, and then to check if all the obtained solutions are somehow *compatible* each other.

But this reduction of the problem to a family of problems corresponding to single gates, only works whenever all the alphabets of the process nodes of the network are the same, and thus for each process-node p and gate a there is some gate-node g labelled with a connected with p. This is the case that we have studied in the second step of our procedure.

Our first idea in order to solve the general case was to try to reduce it to that particular case. But that showed us to be not possible, since the fact that any process appearing in the expressions computed along the application of the algorithm, could eventually execute some action through any of the gates of the network, was crucial in order to prove the correctness of the algorithm.

Therefore we had to change our approach to the question, looking directly for the appropiate generalization of the algorithm to solve the general instance of our problem. If finally we have decided to include in this paper the solution for the previous (partial) case, it is mainly for pedagogical reasons, since both the general algorithm and its correctness proof are absolutely inspired by the corresponding ones for that particular case.

First Case: Networks with a single gate

We consider in this section the particular case in which the system has a single gate, that is to say $|GL(G)| = 1$. In such a case the kind of parallel expressions in which we are interested, can be represented, as already suggested by T. Bolognesi in [2], as arithmetic expressions, rewriting $|||$ into + and $|[a]|$ into $\cdot$.

We can translate the definition of the operational semantics of LOTOS to this arithmetic framework, obtaining the following rules

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1' + E_2} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_1 + E_2'} \qquad \frac{E_1 \xrightarrow{a} E_1' \quad E_2 \xrightarrow{a} E_2'}{E_1 \cdot E_2 \xrightarrow{a} E_1' \cdot E_2'}$$

Besides, the commutativity of both operators and the distributive axiom $(E_1+E_2)\cdot E_3 = (E_1\cdot E_2)+(E_1\cdot E_3)$ are also correct in this framework. Then our problem reduces to prove if that expression can be rewritten into another (equivalent) one with a single occurrence of each process variable, by application of the commutativity of both operators, and of the distributive axiom in the *right to left* way.

In order to check that property, we concentrate on the root operator of the (present state of the) expression to be reduced. If it is a product, then there cannot be any common process variable in its two arguments, and thus the problem can be reduced to the simplification of both arguments. Otherwise we select any process variable with more than a single occurrence in the expression, and we try to elliminate its repeated occurrences by application of the distributive axiom. If it is not possible, then the algorithm *fails*, concluding that the original network is not LOTOS representable. Otherwise we iterate the process until there will be no variable occuring more than one time in the expression. Since there is no necessity of any backtracking along the application of the procedure, it is easy to check that using the adequate data structures to represent the involved expressions, the cost of the algorithm is (in the worst case) cuadratic on the size of the network.

An important auxiliary result, which besides is rather interesting by itself, to prove the correctness of this algorithm, is the one telling us that two essentially different (up to commutativity and associativity) expressions with single appearances of each process variable, are not equivalent. This result could seem to be trivial, what it is somehow disproved by the (relative) complexity of its formal proof. As a matter of fact the correctness proof of our algorithm (which, at least in our opinion, is far from obvious) is nearly immediate once one can use for it this auxiliary result.

Second Case: All the process nodes have a common alphabet

The second step covers the case in which all the process nodes have the same alphabet, which implies that for any process-node p and gate a there is some gate-node g labelled with a and connected with p.

In this case the procedure begins by the application of the algorithm corresponding to the previous case to the projection of the given network over each one of its gates. If any of them is not LOTOS representable, neither the full network is. Otherwise we have to check if all the obtained expressions are *compatible* each other. This means that they can be obtained by projecting over each gate the LOTOS expression that we are searching. Thus we have to check if the *hierarchical relations* between the process variables, induced by the expressions corresponding to each gate of the network, are not *contradictory*.

For we use the fact that for each set of gates A, the corresponding parallel operator $|[A]|$ is associative. Then we can write the expressions involved in the process, in what we call *normal form*, which is formally defined by

Definition 5 a) If $P_1, \ldots, P_n$ are process variables and $A \subseteq Gates$, then the expression $E = |[A]|(P_1, \ldots, P_n)$ is in normal form, and we will say that the set A is its *root synchronization set*, which we will denote by $rss(E)$.
b) If $E_1, \ldots, E_k$ are expressions in normal form, with $rss(E_i) = A_i$, and $A \subseteq Gates$ verifies $A \neq A_i$, for each $i \in \{1, \ldots, k\}$, then the expression $E = |[A]|(E_1, \ldots, E_k)$ is also in normal form, and we take $rss(E) = A$. □

Then, in order to check the consistency of two expressions in normal form, $E = |[A]|(E_1, \ldots, E_m)$ and $F = |[B]|(F_1, \ldots, F_n)$, we first study if its common set of process variables can be partitioned in a family of subsets $P^1, \ldots, P^t$, in such a way that for each $i \in \{1, \ldots, t\}$ either exists some $j \in \{1, \ldots, m\}$ with $Processes(E_j) = P^i$ and some $K \subseteq \{1, \ldots, n\}$ with $\bigcup_{k \in K} Processes(F_k) = P^i$, or there exists some $j \in \{1, \ldots, n\}$ with $Processes(F_j) = P^i$ and some $K \subseteq \{1, \ldots, m\}$ with $\bigcup_{k \in K} Processes(E_k) = P^i$. If this is the case our problem reduces to a family of instances of the same problem, with an instance for each $i \in \{1, \ldots, t\}$; otherwise the given expressions are not consistent. Those instances of the problem are defined in the following way:

- If $\exists j_1 \in \{1, \ldots, m\}\ \exists j_2 \in \{1, \ldots, n\} \mid Processes(E_{j_1}) = Processes(F_{j_2}) = P^i$, we check the consistency of E_{j_1} and F_{j_2}.
- If $\exists j \in \{1, \ldots, m\}\ \exists K \subseteq \{1, \ldots, n\} \mid |K| > 1 \wedge Processes(E_j) = \bigcup_{k \in K} Processes(F_k) = P^i$, we check the consistency of E_{j_1} and $|[B]|_{k \in K}(F_k)$.

- If $\exists j \in \{1, \ldots, n\} \exists K \subseteq \{1, \ldots, m\} \mid |K| > 1 \wedge Processes(F_j) = \bigcup_{k \in K} Processes(E_k) = P^i$, we check the consistency of $|[A]|_{k \in K}(E_k)$ and F_j.

If all these tests are passed then the given expressions are consistent; if this is the case, we also obtain the expression D which combines the structures of both expressions. Otherwise, they are not consistent, and thus they cannot be combined into a single expression.

The cost of this algorithm is cuadratic on the size of the given expressions, and thus the cost of the presented algorithm to test LOTOS-representability in this second case, is less than cubic on the size of the given network.

The General Case

As in the previous case we begin by solving the problems corresponding to each gate in isolation. But in this case, when considering the subproblem corresponding to a gate g, the process nodes which do not contain that gate in its alphabet, are not considered when applying the algorithm described in our first step. Nevertheless, once we have the corresponding solution, we add those processes as new sons of the root of the solution.

It is clear that whenever $|[g]|$ is the operator labelling the root of the expression, we obtain an *impossible synchronization*, since g is not a gate of any of the added processes. But the key idea in order to guarantee the correctness of the algorithm, is that such LOTOS expressions are not interpreted by our algorithm in the ordinary way; instead it uses a modified version of the semantics of the parallel operator, which considers that if we have an expression $E = |[A]|(E_1, \ldots, E_n)$, and we want to execute some $a \in A$, only those processes P_i including a in their alphabets will have to cooperate to do it.

This change implies that the hierarchical structure induced by an expression is no more unique, since the processes which do not contain the corresponding action in its alphabet, could have been added anywhere, and not just below the root of the expression, obtaining a set of different expressions, which however are equivalent with respect to that modified semantics.

This implies that when we check the compatibility of two expressions, we have to let the processes no containing the corresponding gates, to move down into any argument of the corresponding expression, if that is necessary in order to match the structure induced by the other expression.

In order to generalize this idea, to apply it when we have to compare two expressions which have been obtained by composing the solutions corresponding to several gates, we introduce the notion of *independency*, for sets of expressions. We say that the elements of a set of expressions are *independent* iff their alphabets are disjoint each other. Then, whenever we are comparing two such expressions, and we have inside any of them the parallel composition of a family of independent subexpressions, we can put all together obtaining a single subexpression, if that is necessary in order to match the structure induced by the other expression.

Thus, the second step of the algorithm for this general case is obtained from the corresponding step of the algorithm for the previous case, by relaxing the condition which imposes that for each set P^i in the considered partition, we have either to find some single subexpression E_j (or F_j) with $Processes(E_j) = P^i$ (or equivalently for F_j), allowing now to have instead a family $\{E_j \mid j \in J\}$ (or equivalently for F's), of independent processes. When we use that possibility, and for each $j \in J$ we have $E_j = |[A_j]|(E_{j1}, \ldots, E_{jm_j})$ then the expression to be compared in the instance of the problem corresponding to P^i, will be that given by $|[A_j]|_{j \in J, k \in \{1, \ldots, m_j\}}(E_{jk})$.

Finally, if the algorithm succeeds, those impossible synchronizations remaining in the obtained solution are removed, since they were only added as a consequence of a technical trick, to obtain an algorithm as similar as possible to that corresponding to the second case. By removing them we recover the ordinary semantics for the parallel operator, and thus the obtained expression is indeed the representation of the given network.

References

[1] T. Bolognesi *A Graphical Composition Theorem for Networks of LOTOS Processes* in Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE Computer Society Press, 1990.

[2] T. Bolognesi (Ed.) *Catalogue of LOTOS Correctness Preserving Transformations* LOTOSPHERE Task 1.2 Final Deliverable Lo/WP1/T1.2/N0045 ESPRIT 2304: LOTOSPHERE (1992)

[3] T. Bolognesi, D. de Frutos, Y.Ortega *Graphical Composition Theorems for Parallel and Hiding Operators* in Formal Description Tecniques III, (Procs. FORTE'90) North Holland (1991)

[4] J. Hinterplattner, H. Nirschl, H. Saria *Process Topology Diagrams* in Formal Description Tecniques III, (Procs. FORTE'90) North Holland (1991)

Towards Performance Evaluation in Process Algebras [*]

Roberto Gorrieri Marco Roccetti

Università di Bologna, Dipartimento di Matematica
Piazza di Porta San Donato 5, I - 40127 Bologna (Italy)
e-mail:{gorrieri,roccetti}@cs.unibo.it

Process algebras, such as ACP, CCS, CSP and LOTOS, are widely accepted formalisms for the "functional" specification of concurrent systems, where functional means that a process term specifies *what* actions the system should do. Bisimulation is a standard tool for the definition of a behavioural equivalence on process terms which, besides the actions, considers the structure of the alternative choices (branching-time semantics).

Another, not less relevant, aspect of a system specification is its "performance", i.e., the measure of the time consumed for execution. It may be argued that performance is only a matter of efficient implementation. This is untrue: For applications whose functionality is performance-dependent (i.e., it can be altered by the flow of time like, e.g., in presence of time-outs), it is reasonable to require that a specification does not allow implementations which do not have an adequate performance.

Our work gives a contribution in the direction of integrating the two needs by presenting a new bisimulation-based semantics, called *performance equivalence*, for a simple process algebra where systems are equated if they perform the same actions in the same time (i.e., they have the same functional and performance behaviour).

The basic assumptions on which this semantics relies are the following. Any action a has a duration — a natural number $f(a)$ — which represents the time units needed for its execution. Every sequential subsystem is equipped with a clock, whose elapsing is set only by the execution of actions. To be more precise, whenever an action a is executed by a sequential subcomponent P, the value n of the local clock of P is incremented to $n + f(a)$, whilst the local clocks of those sequential components not involved in the execution of a are unaffected. Hence, if P is idle during a transition, its local clock value cannot increase. In other words, each sequential subsystem is always *eager* to perform an executable action (or dually "actions are urgent"): the time value is incremented locally only when the executable action is performed. The only exception is concerned with synchronization. Two processes can synchronize when they perform the same action at the same time; if one of the two is able to execute such an action before the other one, then a form of "busy waiting" is allowed. This fact shows that the local clocks are indeed locally replicated, possibly inconsistent, versions of the unique physical global time. Indeed, the time is the same for all the sequential components; the only point is that we do not pretend that all the local views be consistent during the simulation. This assumption is rather natural if we are interested in performance evaluation only. In a simulation there is no need of having a tight agreement between the *time of execution* (i.e., the number attached to the executed actions) and the *time of observation* (i.e., the time of "generation" of the action during the simulation).

A simple example may be helpful in clarifying the basic idea. Consider the term $E = a.g.nil \parallel b.nil$. Since the clock is set to 0 before starting the execution of E, the initial state of the transition system is $(0 \Rightarrow a.g.nil) \parallel (0 \Rightarrow b.nil)$, where the auxiliary operator $n \Rightarrow P$ means that

[*] This work has been partially supported by Esprit Basic Research project BROADCAST n.6360 and by Italian CNR, grant n. 92.00069.CT12.115.25585.

the execution of P starts exactly after n time units of the global clock. One of the two transitions out of it is labelled $\langle a, f(a)\rangle$ and reaches $(f(a) \Rightarrow g.nil) \parallel 0 \Rightarrow b.nil$. By executing b, we reach the state $f(a) \Rightarrow g.nil \parallel f(b) \Rightarrow nil$; finally the execution of g produces a transition labelled $\langle g, f(a) + f(g)\rangle$ with target state $(f(a) + f(g) \Rightarrow nil) \parallel f(b) \Rightarrow nil$. It is immediate observing that the time needed for the complete execution of the system is $max\{f(a) + f(g), f(b)\}$ and that bisimulation equivalence over this labelled transition system is more discriminating than interleaving bisimulation. Indeed, the equation $a.nil \parallel b.nil = a.b.nil + b.a.nil$ does not hold

$$0 \Rightarrow a.nil \parallel 0 \Rightarrow b.nil \xrightarrow{\langle a,f(a)\rangle} f(a) \Rightarrow nil \parallel 0 \Rightarrow b.nil \xrightarrow{\langle b,f(b)\rangle} f(a) \Rightarrow nil \parallel f(b) \Rightarrow nil$$

$$0 \Rightarrow a.b.nil + b.a.nil \xrightarrow{\langle a,f(a)\rangle} f(a) \Rightarrow b.nil \xrightarrow{\langle b,f(a)+f(b)\rangle} f(a) + f(b) \Rightarrow nil$$

as the execution of b after a in the left-hand-side term is performed with a higher clock value. Notice that, if $f(a) > f(b)$, then the execution of a before b in $a.nil \parallel b.nil$ generates two transitions where the clock value is decreased in the second transition. This phenomenon has been criticized in real-time literature (e.g., [2]), because in this context the time of execution and the time of observation are required to agree tightly; however, a recent report [1] shows that this view can be reasonably accepted also in real-time applications, provided that those "ill-timed" traces are "well-caused", as we do here.

The simple process algebra $\mathcal{L}$ we study has operators borrowed from CCS and TCSP:

$$E ::= nil \mid a.E \mid E + E \mid E \parallel_A E \mid E[\Phi]$$

where, for the sake of simplicity, we assume that Φ relabels actions with the same duration, i.e., $f(a) = f(\Phi(a))$ must hold for any a. $\mathcal{L}$ is equipped with an SOS semantics in terms of labelled transition systems. The states are terms generated by the following syntax:

$$s ::= n \Rightarrow nil \mid n \Rightarrow a.E \mid s + s \mid s \parallel_A s \mid s[\Phi]$$

where E denotes any (finite) $\mathcal{L}$ term. When prefixing a term E with a *clock value* n, $n \Rightarrow E$, we mean that n distributes over the operators, till to the sequential components. Formally:

$$\begin{aligned} n \Rightarrow (E + E') &= (n \Rightarrow E) + (n \Rightarrow E') \\ n \Rightarrow (E \parallel_A E') &= (n \Rightarrow E) \parallel_A (n \Rightarrow E') \\ n \Rightarrow (E[\Phi]) &= (n \Rightarrow E)[\Phi] \end{aligned}$$

Hence, $n \Rightarrow E$ is (canonically) reduced to a state. Each transition is labelled by $\langle a, n\rangle \star \omega$: the *observable part* is $\langle a, n\rangle$, meaning that action a has been completed exactly after n time units, while the *location part* ω is a term pointing out which sequential subagents have been involved in the execution of action a itself. The latter part, irrelevant from an observational viewpoint (and thus omitted in the previous examples), is used to guarantee a correct updating of the local clock values in steps of synchronization. Locations are generated as follows:

$$\omega ::= \bullet \mid \omega\rfloor \mid \lfloor\omega \mid \omega \parallel \omega$$

where $\bullet$ (one place) means "sequential", while $\omega\rfloor$ ($\lfloor\omega$) means that the system is composed of two main parts and that ω comes from the *left* (*right*) part; the operator $\parallel$ is a form of (disjoint) union of locations. The rules for prefixing, sum, relabelling and parallel composition (where the obvious symmetric rules are omitted) are:

$$\frac{}{n \Rightarrow a.E \xrightarrow{\langle a,k\rangle\star\bullet} k \Rightarrow E}\; k = n + f(a) \qquad\qquad \frac{s \xrightarrow{\langle a,n\rangle\star\omega} s'}{s[\Phi] \xrightarrow{\langle\Phi(a),n\rangle\star\omega} s'[\Phi]}$$

$$\frac{s_1 \xrightarrow{(a,n)\star\omega} s_1'}{s_1 + s_2 \xrightarrow{(a,n)\star\omega} s_1'} \qquad\qquad \frac{s_1 \xrightarrow{(a,n)\star\omega} s_1'}{s_1 \parallel_A s_2 \xrightarrow{(a,n)\star(\omega\rfloor)} s_1' \parallel_A s_2} \text{ if } a \notin A$$

$$\frac{s_1 \xrightarrow{(a,n_1)\star\omega_1} s_1' \quad s_2 \xrightarrow{(a,n_2)\star\omega_2} s_2'}{s_1 \parallel_A s_2 \xrightarrow{(a,n_1)\star(\omega_1\parallel\omega_2)} s_1' \parallel_A ([n_1,\omega_2]s_2')} \text{ if } a \in A \text{ and } n_1 \geq n_2$$

where the auxiliary *time-update* operator $[n_1,\omega_2]$ applied to s_2' in the synchronisation rule increases to n_1 the clock value of the sequential components of s_2' singled out by ω_2. Indeed, two sequential subsystems can synchronize via the same action a performed during the same time interval; hence, if one of the two is ready before the other one, it must wait. Formally, some of the equations defining the time-update operator are:

$$\begin{aligned} [n,\bullet](m \Rightarrow a.E) &= n \Rightarrow a.E \\ [n,\omega\rfloor](s_1 \parallel_A s_2) &= ([n,\omega]s_1) \parallel_A s_2 \\ [n,\omega_1 \parallel \omega_2](s_1 \parallel_A s_2) &= ([n,\omega_1]s_1) \parallel_A ([n,\omega_2]s_2) \end{aligned}$$

Finally, we introduce an auxiliary operator which forgets about the additional location part on the labels. The "forgetful" operator, $F(s)$, is defined as follows:

$$\frac{s \xrightarrow{(a,n)\star\omega} s'}{F(s) \xrightarrow{(a,n)} F(s')}$$

We say that two $\mathcal{L}$ terms E_1 and E_2 are *performance equivalent*, $E_1 \sim_p E_2$, if and only if $F(0 \Rightarrow E_1)$ is bisimilar to $F(0 \Rightarrow E_2)$.

Performance equivalence is preserved by all the operators, except parallel composition (the counter-example is the same proposed by A. Rabinovich to show that partial ordering bisimulation is not a congruence). A complete proof system for performance equivalence can be easily provided with the help of some auxiliary operators. Two terms E_1 and E_2 are performance equivalent if and only if $F(0 \Rightarrow E_1)$ is proved equal to $F(0 \Rightarrow E_2)$, according to the following axioms. Besides the usual laws for bisimulation (+ is a commuative, nil absorbent monoid), the distributive axioms on states and the axioms for the time-update operator listed above, there are axioms which transform each state in a tree labelled also with the location part, and some more axioms that, taken such a tree, forgets this location part. Some of them are reported below:

$$n \Rightarrow a.E = (\langle a,k\rangle \star \bullet).(k \Rightarrow E) \quad \text{for } k = n + f(a)$$

$$s_1 \parallel_A s_2 = s_1 \rfloor_A s_2 + s_1 \lfloor_A s_2 + s_1 \mid_A s_2$$

$$(\langle a,n\rangle \star \omega.s_1)\rfloor_A s_2 = (\langle a,n\rangle \star \omega\rfloor).(s_1 \parallel_A s_2) \quad \text{if } a \notin A$$

$$F(\langle a,n\rangle \star \omega.s) = \langle a,n\rangle.F(s)$$

Hence, the procedure for checking if two terms are performance bisimilar is as follows: first, generate a tree out of a state; second, forget the location part; third, check their equivalence according to the axioms of strong bisimulation.

Performance equivalence is a non-interleaving semantics which is based on the notion of time-consumption. It is interesting to see what is the rank of this equivalence in the large spectrum of non-interleaving semantics proposed in the literature. It can be proved that partial ordering bisimulation [4], $\sim_{po}$, is finer than $\sim_p$. This is quite obvious as causality gives enough information to recover the time needed for execution. The reverse does not hold in general; however, it can be proved that if the two systems are *time-deterministic* (there are no reachable states with two outgoing transitions labelled by the same action at the same time), then performance equivalence induces a semantics which is even finer than $\sim_{po}$. Let us consider ST semantics [7]. Performance equivalence is strictly finer than ST bisimulation semantics. The counterexample is easy: consider the two terms $a.nil \parallel b.nil$ and $((a.c.nil \parallel_c (b.nil + c.nil))[{}^{b}/_{c}]$. These are ST bisimilar but not performance bisimilar, as in the latter system, action b can be completed also after $f(a) + f(b)$ time units. This example may also help in clarifying in which sense our semantics is not "real-time", according to some papers in the literature (see, e.g., [8]). A different operational semantics for synchronisation, which is claimed "real-time", requires that both agents are ready to perform the same action at the same time, without any busy-waiting. This solution, proposed in, e.g., [1], forbids several executions that we prefer to keep; for instance, the synchronisation over c in the example above. Indeed, our treatment of nondeterminism is similar to an internal choice: first, each local component decides (with zero delay) which action it wants to execute, then it tries to export the action to the top level, possibly delayed by synchronisations. If successful, the execution takes exactly the right amount of time; otherwise, the action is not executed at all. As all the local choices are to be taken into account, all the possible executions are represented. (Our view is shared by other researchers, proposing similar ideas in different contexts [3, 5].)

This semantics is *a priori* of timed calculi (no specific operators have been proposed to this aim) and real-time is not an issue of this paper. Nonetheless, we feel that our approach to time in system executions can be helpful for a formal description of time-dependent programming constructs such as timeout, watchdog, and so on. In conclusion, the aim of our study is to provide an approach able to incorporate time into formal specifications, in order to capture functional and performance behaviour of distributed and parallel systems. Nevertheless we are aware that, because of the ineherent random nature of the investigated problems, the concepts of random variables and stochastic processes represent the unique well-founded discipline able to describe performance aspects of computer systems. Thus, even if other alternative (or complementary) approaches can be studied (e. g. Stochastic Petri Net models), our next purpose will be to replace specific, deterministic time duration values with time probabilistic distribution duration functions, in order to provide a uniform integration of the theories of process algebras and performance evaluation (see, e.g., [6] for a preliminary study). This justifies the title of the paper.

Acknowledgements

We are grateful to Luca Aceto and the anonymous referees for their useful comments.

References

[1] L. Aceto, D. Murphy, *On the Ill-timed but Well-caused*, to appear in Proc. CONCUR'93, LNCS, Springer, 1993.

[2] J. Baeten, J.A. Bergstra, *Real Time Process Algebra*, Formal Aspects of Computing 3(2):142–188, 1991.

[3] E. Best, *Weighted basic Petri Nets*, in Concurrency'88, LNCS 335, Springer, 257–276, 1988.

[4] P. Degano, R. De Nicola, U. Montanari, *Observational Equivalences for Concurrency Models*, in Formal Description of Programming Concepts III, North-Holland, 105–132, 1987.

[5] G.L. Ferrari, U. Montanari, *Observing Time-Complexity of Concurrent Programs*, submitted for publication.

[6] N. Gotz, U. Herzog, M. Rettelbach, *TIPP- Introduction and Application to Protocol Performance Analysis*, to appear in Tutorial Proceedings of Performance '93, October 1993.

[7] R.J. van Glabbeek, F. Vaandrager, *Petri Net Models for Algebraic Theories of Concurrency*, in Proc. PARLE II, LNCS 259, Springer, 224-242, 1987.

[8] W. Vogler, *Timed Testing of Concurrent Systems*, to appear in Proc. ICALP'93, LNCS, Springer, July 1993.

Translation Results for Modal Logics of Reactive Systems

F. Laroussinie[*], S. Pinchinat[†] and Ph. Schnoebelen[*]
LIFIA-IMAG, Grenoble, France
and University of Sussex, Brighton, UK

Modal logics are an important tool in the analysis, specification and verification of reactive systems. Among many other applications, logics like HML have been used as a benchmark for semantic equivalences [8], as the specification language used in model checking tools [1], and as a language in which to explain why two systems are not semantically equivalent [10].

Regarding modal characterizations of semantic equivalences, the classical result is the adequacy theorem of Hennessy and Milner who showed that two states in a (finitely branching) transition system are bisimilar, written $p \underline{\leftrightarrow} q$, iff they satisfy the same HML formulas, written $p \equiv_{HML} q$, where $p \equiv_L q \overset{\text{def}}{\Leftrightarrow} \forall f \in L\ (p \models f \ \Leftrightarrow\ q \models f)$.

Here we are mostly interested in modal logics with past-time (backward) operators. A few exist. They have been used (among other applications) to capture non-continuous properties of *generalized* transitions systems (J_T in [9]), to capture history-preserving bisimulation in causality-based models (L_P in [2]) and to capture *branching bisimulation* by mimicking back-and-forth τ-bisimulation (L_B in [5]).

In this paper we give three non-trivial translation theorems of the generic form $L \preceq L'$ showing, given any formula f from some modal logic L, how to build an equivalent $f' \in L'$. This kind of problem has not received much attention in modal logics of reactive systems and the existing results in temporal logics mostly deal with future-time logics.

Our translations are defined by rewrite rules (to apply with a given strategy) over formulas. A consequence is that, once discovered, the translations are easy to implement. Our motivations are not only theoretical. For example, by showing how to translate HML_{bf} (HML with past-time connectives) into its future-time fragment HML, we show how to easily expand the input language of any software tool (e.g. a verifier) handling HML properties.

1 Backward modalities

We consider a fixed set $A = \{a, b, \ldots\}$ of *labels*. A *labeled transition system* (LTS) is an edge-labeled graph $\langle Q, \rightarrow \rangle$ where $Q = \{p, q, \ldots\}$ is a set of *states* and $\rightarrow \subseteq Q \times A \times Q$ is the *transition relation*. We assume a fixed LTS S.

HML_{bf} is HML with past-tense modalities[1] and has the following grammar:

$$HML_{bf} \ni f, g \ ::= \ \top \mid \neg f \mid f \wedge g \mid \langle a \rangle f \mid \overline{\langle a \rangle} f$$

where a is any action from A. (HML is the fragment of HML_{bf} where the $\overline{\langle a \rangle}$ operators are not allowed.) $f, g, \alpha, \beta, \varphi, \psi, \ldots$ denote HML formulas and we use the standard abbreviations: $f \vee g$, $\perp$, $[a]f$ for $\neg \langle a \rangle \neg f$, ...

A modal logic with backward modalities states properties of a *run* $\pi = [q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_n} q_n]$ of S, that is, of state q_n with a given history (or past). We write $\pi \xrightarrow{a} \pi'$ when run π' is π with a transition $q_n \xrightarrow{a} q_{n+1}$ added. For a run π and an HML_{bf} formula f, we define $\pi \models f$ by induction on the structure of f:

$$\begin{aligned} \pi \models \langle a \rangle f & \quad \text{iff there is a } \pi \xrightarrow{a} \pi' \text{ s.t. } \pi' \models f, \\ \pi \models \overline{\langle a \rangle} f & \quad \text{iff there is a } \pi' \xrightarrow{a} \pi \text{ s.t. } \pi' \models f. \end{aligned}$$

(the other clauses are obvious.) Then, for a state $q \in Q$, the derived notion $q \models f$ is given by $q \models f \overset{\text{def}}{\Leftrightarrow} [q] \models f$.

[5] mention that $p \equiv_{HML_{bf}} q$ iff $p \underline{\leftrightarrow} q$ because (strong) bisimulation coincides with (strong) back-and-forth bisimulation [3]. This entails $p \equiv_{HML} q$ iff $p \equiv_{HML_{bf}} q$.

We are looking for a more detailed comparison of the expressive power of HML and HML_{bf}. We consider whether formulas of HML_{bf} can be translated into HML. This requires some definitions:

Definition 1 *Two formulas are* globally equivalent, *written $f \equiv_g f'$, iff $\pi \models f \ \Leftrightarrow\ \pi \models f'$ for all runs in all LTS's.*

They are initially equivalent, *written $f \equiv_i f'$, iff $q \models f \Leftrightarrow q \models f'$ for all states in all LTS's.*

Clearly, $f \equiv_g f'$ implies $f \equiv_i f'$ but the converse is not true. $\equiv_g$ is a congruence: when $f \equiv_g f'$ with f a subformula of g then $g \equiv_g g[f'/f]$. This does not hold for $\equiv_i$ which is only a congruence w.r.t. boolean contexts.

[*] LIFIA-IMAG, 46 Av. Félix Viallet, F-38000 Grenoble, FRANCE. Email:{francois,phs}@lifia.imag.fr

[†] School of Computing & Cognitive Sciences, University of Sussex, Falmer, Brighton, BN1 9QH, UK. Email:sp@cogs.sussex.ac.uk. This author is supported by an INRIA grant.

[1] It was introduced in [5] for systems with τ's (but note that HML_{bf} is a subset of J_T defined in [9].)

Definition 2 *A logic L can be translated (resp. initially translated) into L', written $L \preceq_g L'$ (resp. $L \preceq_i L'$), iff for any $f \in L$ there is a $f' \in L'$ with $f \equiv_g f'$ (resp. $f \equiv_i f'$).*

$L \preceq_i L'$ implies $\equiv_{L'} \subseteq \equiv_L$ but the converse is not true in general.

Theorem 1 $HML_{bf} \preceq_i HML$.

The proof uses three steps.
• Say a formula is *restricted* if it has the form $\langle a\rangle f$, $\neg\langle a\rangle f$, $\overline{\langle a\rangle} f$ or $\neg\overline{\langle a\rangle} f$, with f a *restricted conjunct*, i.e. a (possibly empty) conjunction of restricted formulas. (We use $\varphi, \psi, \ldots$ to denote restricted formulas.) Then

Lemma 1 *Any f is equivalent to a disjunction of restricted conjuncts.*

• Say a formula is *separated* if no backward modality occurs in the scope of a forward modality [2] (and write HML_{bf}^{sep} for the fragment of HML_{bf} that contains only separated formulas).

Lemma 2 *Any restricted φ is equivalent to a separated formula.*

Proof Rewrite φ into a separated formula using:

$$\langle a\rangle(\psi \wedge \overline{\langle b\rangle}\varphi) \quad \equiv_g \quad \begin{cases} \bot & \text{if } a \neq b, \\ \varphi \wedge \langle a\rangle\psi & \text{if } a = b. \end{cases}$$

$$\langle a\rangle(\psi \wedge \neg\overline{\langle b\rangle}\varphi) \quad \equiv_g \quad \begin{cases} \langle a\rangle\psi & \text{if } a \neq b, \\ \neg\varphi \wedge \langle a\rangle\psi & \text{if } a = b. \end{cases}$$

Any restricted non-separated formula can be rewritten according to one of these equations. Applying an outermost strategy guarantees that non-separated subformulas remain restricted conjuncts. Termination is clear. □

Proposition 1 *(Separation Lemma)*

$$HML_{bf} \preceq_g HML_{bf}^{sep} \tag{1}$$

is the immediate corollary. (Observe that (1) does not hold for Gabbay's definition of separated formulas.)

Now we conclude the proof of Theorem 1 with

Proposition 2 $HML_{bf}^{sep} \preceq_i HML$.

Proof Use $\overline{\langle a\rangle} f \equiv_i \bot$ to eliminate (modulo $\equiv_i$) any past-time modality which is not in the scope of a future-time modality. □

2 τ-moves, from L_U to L_{BF}

For transition systems labeled over $A_\tau \stackrel{\text{def}}{=} A \cup \{\tau\}$, [5] introduces L_U and L_{BF}, two modal logics characterizing branching bisimulation.

L_{BF} is a version of HML_{bf} adapted to systems with silent moves. Its grammar is

$$L_{BF} \ni f, g \ ::= \ \top \mid \neg f \mid f \wedge g \mid \langle\langle k\rangle\rangle f \mid \overline{\langle\langle k\rangle\rangle} f$$

where k is any label from $A_\epsilon \stackrel{\text{def}}{=} A \cup \{\epsilon\}$. We use $[[k]]f$ and $\overline{[[k]]}f$ as standard abbreviations. The semantics of the new modalities is given by:

$$\begin{array}{ll} \pi \models \langle\langle a\rangle\rangle f & \text{iff there is a } \pi \stackrel{\epsilon}{\Rightarrow}\xrightarrow{a}\stackrel{\epsilon}{\Rightarrow} \pi' \text{ s.t. } \pi' \models f, \\ \pi \models \langle\langle \epsilon\rangle\rangle f & \text{iff there is a } \pi \stackrel{\epsilon}{\Rightarrow} \pi' \text{ s.t. } \pi' \models f. \end{array}$$

where $\stackrel{\epsilon}{\Rightarrow}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$. The clauses for $\overline{\langle\langle k\rangle\rangle}$ are just like for $\langle\langle k\rangle\rangle$, only backward.

L_U has no backward modalities but it has a so-called "until" operator which is more powerful that the simple future-time operator of L_{BF}. The grammar of L_U is

$$L_U \ni f, g \ ::= \ \top \mid \neg f \mid f \wedge g \mid f\langle k\rangle g$$

with $k \in A_\epsilon$. The semantics is given by

$$\begin{array}{ll} \pi \models f\langle a\rangle g & \text{iff } \exists n > 0,\ \pi = \pi_0 \xrightarrow{\tau} \pi_1 \xrightarrow{\tau} \cdots \pi_{n-1} \xrightarrow{a} \pi_n \\ & \quad \text{s.t. } \pi_n \models g \text{ and } \pi_i \models f \text{ for } i < n, \\ \pi \models f\langle \epsilon\rangle g & \text{iff } \exists n \geq 0,\ \pi = \pi_0 \xrightarrow{\tau} \pi_1 \xrightarrow{\tau} \cdots \pi_{n-1} \xrightarrow{\tau} \pi_n \\ & \quad \text{s.t. } \pi_n \models g \text{ and } \pi_i \models f \text{ for } i < n. \end{array}$$

For technical reasons, we introduce L_{BU} [11], a logic built by combining all modalities of L_U and of L_{BF}, so that both L_{BF} and L_U are fragments of a common superset:

$$L_{BU} \ni f, g \ ::= \ \top \mid \neg f \mid f \wedge g \mid \langle\langle k\rangle\rangle f \mid \overline{\langle\langle k\rangle\rangle} f \mid f\langle k\rangle g$$

with $k \in A_\epsilon$. In L_{BU}, the $\langle\langle k\rangle\rangle$ is not really needed because

$$\langle\langle k\rangle\rangle f \equiv_g \top\langle k\rangle(\top\langle\epsilon\rangle f) \tag{2}$$

Considering that $\equiv_{L_U}$ and $\equiv_{L_{BF}}$ coincide [5], a natural question is whether L_U or L_{BF} can be translated into the other. At a certain time, the authors of [5] tried to simply embed L_U into L_{BF} (see Theorem 2.19 in [4]) but later found a mistake in their proof. A translation exists but it is not trivial:

Theorem 2 $L_U \preceq_g L_{BF}$.

Proof We show how to eliminate the until modalities from an L_U formula f.
• Consider a formula f having the general form:

$$f = \Big(\bigvee_{i=1\ldots n} \Big(\bigwedge_{j \in J_i} [[k_{ij}]]\varphi_{ij} \ \wedge \ \bigwedge_{j \in J'_i} \langle\langle k'_{ij}\rangle\rangle \varphi'_{ij} \Big) \Big) \langle k\rangle\, \psi \tag{3}$$

[2] Note that [6, 7] use a different, less general, definition of separated formulas: say a formula is *pure-future* if it does not contain past-time operator, is *pure-past* if it does not contain future-time operator, and is *separated (in Gabbay's sense)* if it is a boolean combination of pure-past and pure-future formulas.

for which we introduce the following simplifying abbreviations:

$$[\alpha_i] \stackrel{\text{def}}{=} \bigwedge_{j\in J_i} [[k_{ij}]]\varphi_{ij} \qquad\qquad (\alpha'_i) \stackrel{\text{def}}{=} \bigwedge_{j\in J'_i} \langle\langle k'_{ij}\rangle\rangle\varphi'_{ij}$$

The top modality of f is an until with a n-ary disjunction in the left-hand side.

• First, consider the simpler case where $n = 1$ in f. Then if $k = a \in A$ we have

$$f \equiv_s [\alpha_1] \;\wedge\; \langle\langle a\rangle\rangle\Big(\overline{[[\epsilon]]}\psi \wedge \overline{[[a]]}(\alpha'_1)\Big)$$

while if $k = \epsilon$ we have

$$f \equiv_s \psi \vee \Bigg([\alpha_1] \wedge \langle\langle\epsilon\rangle\rangle\Big(\psi \wedge \overline{[[\epsilon]]}(\psi \vee (\alpha'_1))\Big)\Bigg)$$

• Now in the general n-ary case with $n > 1$, we show how to rewrite (3) into a formula where the until is eliminated by introducing new until-formulas having $n-1$-ary disjunctions in their left-hand sides.

If $k = a \in A$ we have

$$f \equiv_s \bigvee_{i=1\ldots n}\Bigg([\alpha_i] \wedge \left(\begin{array}{c} \langle\langle a\rangle\rangle(\overline{[[\epsilon]]}\psi \wedge \overline{[[a]]}(\alpha'_i)) \\ \vee\, \langle\langle\epsilon\rangle\rangle(\psi_i \wedge \overline{[[\epsilon]]}(\psi_i \vee (\alpha'_i))) \end{array}\right)\Bigg)$$

where

$$\psi_i \stackrel{\text{def}}{=} \Big(\bigvee_{\substack{k=1\ldots n\\ k\neq i}} [\alpha_k] \wedge (\alpha'_k)\Big)\langle\epsilon\rangle\psi \qquad\qquad \text{(for any } i \in I)$$

are the new until-formulas containing only $n-1$ members in the disjunction.

Similarly, if $k = \epsilon$, we have

$$f \equiv_s \psi \vee \bigvee_{i=1\ldots n}\Big([\alpha_i] \wedge \langle\langle\epsilon\rangle\rangle\Big(\psi_i \wedge \overline{[[\epsilon]]}(\psi_i \vee (\alpha'_i))\Big)\Big)$$

• Now, with a sound strategy, these two transformations can be used to rewrite an arbitrary f from L_U into L_{BF}:

1. Observe that f in (3) is a quite general until formula except that it has no backward combinator (immediately) in the left-hand side of the until. Say an *FB-formula* is any L_{BU} formula where (i) no until is in the scope of a backward modality, and (ii) where every backward modality is (immediately, but disregarding boolean combinators) under a forward L_{BF} modality.
2. Then if f in (3) is an FB-formula, our transformations give in all cases a formula equivalent to f which is still FB: all the backward combinators we introduce have no until in their scope and are immediately under a forward L_{BF} modality.
3. Now given any formula in L_U, we just have to work by picking the innermost untils first and by writing their left-hand sides in disjunctive normal form. We eventually obtain an L_{BF} formula.

□

3 From L_{BF} to L_U

Theorem 3 *$L_{BF} \preceq_i L_U$.*

This problem was considered in [11] where a partial solution is proposed. Our approach was developed independently and uses our separation techniques. Write L_{BU}^{sep} for the set of separated L_{BU} formulas, i.e. of formulas with no backward modality under the scope of a forward (or until) modality. We show how to rewrite any L_{BU} formula into an equivalent separated formula. The most difficult part here is to find a strategy which ensures termination. For this we use an approach inspired from [6].

Lemma 3 *Any L_{BU} formula f with only one subformula of the form $\overline{\langle\langle k\rangle\rangle}\psi$, where ψ has no modality, is equivalent to a separated formula. (Note: f may contain several occurrences of $\overline{\langle\langle k\rangle\rangle}\psi$.)*

The basic transformation removes a modality $\overline{\langle\langle k\rangle\rangle}$ from the scope of an until modality. First of all, we need not consider disjunctions in the right-hand side of an until because

$$\varphi\langle k\rangle(\psi_1 \vee \psi_2) \equiv_s \varphi\langle k\rangle\psi_1 \vee \varphi\langle k\rangle\psi_2$$

Then conjunctions in the right-hand side can be dealt with by using

$$\alpha\langle a\rangle\big(\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \beta\big) \quad \equiv_s \alpha\langle a\rangle(\psi \wedge \beta)$$

$$\alpha\langle a\rangle\big(\neg\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \beta\big) \quad \equiv_s \alpha\langle a\rangle(\neg\psi \wedge \beta)$$

$$\begin{array}{lll} \alpha\langle a\rangle\big(\overline{\langle\langle b\rangle\rangle}\psi \wedge \beta\big) & \equiv_s \bot & \text{if } a \neq b, \\ & \equiv_s \big(\alpha\langle\epsilon\rangle(\psi \wedge \alpha\langle a\rangle\beta)\big) \vee \big(\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \alpha\langle a\rangle\beta\big) & \text{if } a = b. \end{array}$$

$$\begin{array}{lll} \alpha\langle a\rangle\big(\neg\overline{\langle\langle b\rangle\rangle}\psi \wedge \beta\big) & \equiv_s \alpha\langle a\rangle\beta & \text{if } a \neq b, \\ & \equiv_s \neg\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge (\alpha \wedge \neg\psi)\langle a\rangle\beta & \text{if } a = b. \end{array}$$

$$\begin{array}{l} \alpha\langle\epsilon\rangle\big(\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \beta\big) \\ \qquad\qquad \equiv_s \big(\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \alpha\langle\epsilon\rangle\beta\big) \vee \alpha\langle\epsilon\rangle(\psi \wedge \alpha\langle\epsilon\rangle\beta) \end{array}$$

$$\begin{array}{l} \alpha\langle\epsilon\rangle\big(\neg\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \beta\big) \\ \qquad\qquad \equiv_s \neg\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge (\alpha \wedge \neg\psi)\langle\epsilon\rangle(\beta \wedge \neg\psi) \end{array}$$

$$\alpha\langle\epsilon\rangle\big(\overline{\langle\langle b\rangle\rangle}\psi \wedge \beta\big) \quad \equiv_s \overline{\langle\langle b\rangle\rangle}\psi \wedge \alpha\langle\epsilon\rangle\beta$$

$$\alpha\langle\epsilon\rangle\big(\neg\overline{\langle\langle b\rangle\rangle}\psi \wedge \beta\big) \quad \equiv_s \neg\overline{\langle\langle b\rangle\rangle}\psi \wedge \alpha\langle\epsilon\rangle\beta$$

which are correct without any hypothesis on α, β and ψ.

To remove $\overline{\langle\langle k\rangle\rangle}\psi$ in the left-hand sides of until-formulas, we only consider the general form:

$$\Big((\overline{\langle\langle k\rangle\rangle}\psi \wedge \varphi) \vee (\neg\overline{\langle\langle k\rangle\rangle}\psi \wedge \varphi') \vee \beta\Big)\langle k'\rangle\alpha \qquad (4)$$

We use

$$\Big((\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \varphi) \vee (\neg\overline{\langle\langle\epsilon\rangle\rangle}\psi \wedge \varphi') \vee \beta \Big] (k)\alpha$$
$$\begin{aligned}
&\equiv_s \langle\langle\epsilon\rangle\rangle\psi \wedge (\varphi \vee \beta)(k)\alpha \\
&\quad \vee \; \neg\langle\langle\epsilon\rangle\rangle\psi \wedge \big(\neg\psi \wedge (\varphi' \vee \beta)\big)(k)\alpha \\
&\quad \vee \; \neg\langle\langle\epsilon\rangle\rangle\psi \wedge \big(\neg\psi \wedge (\varphi' \vee \beta)\big)(\epsilon)\big(\psi \wedge (\varphi \vee \beta)(k)\alpha\big)
\end{aligned}$$

$$\Big((\overline{\langle\langle b\rangle\rangle}\psi \wedge \varphi) \vee (\neg\overline{\langle\langle b\rangle\rangle}\psi \wedge \varphi') \vee \beta \Big] (k)\alpha$$
$$\equiv_s \Big(\overline{\langle\langle b\rangle\rangle}\psi \wedge (\varphi \vee \beta)(k)\alpha\Big) \vee \Big(\neg\overline{\langle\langle b\rangle\rangle}\psi \wedge (\varphi' \vee \beta)(k)\alpha\Big)$$

We have no room here to show the rules for the general cases where $\overline{\langle\langle k\rangle\rangle}\psi$ occurs in both sides of the until. They are often dealt with by a combination of the previous transformations, and in some cases by new transformations in the same spirit.

Once this basic step is established we just have to offer a strategy ensuring termination:

Lemma 4 *Any L_{BU} formula f with only n subformulas of the form $\overline{\langle\langle k_i\rangle\rangle}\psi_i$, where ψ_i has no modality, is equivalent to a separated formula.*

Proof Use Lemma 3 on each $\overline{\langle\langle k_i\rangle\rangle}\psi_i$ in turn. □

Lemma 5 *Any L_{BU} formula f with only n subformulas of the form $\overline{\langle\langle k_i\rangle\rangle}\psi_i$, where ψ_i has only backward modalities, is equivalent to a separated formula.*

Proof Use Lemma 3 to extract the $\overline{\langle\langle k_i\rangle\rangle}\psi_i$. This may introduce new $\overline{\langle\langle k_{ij}\rangle\rangle}\psi_{ij}$ in the (immediate) scope of some untils, but these were subformulas of the ψ_i so that the height of the maximal nesting of backward modalities is decreased. □

Lemma 6 *Any L_{BU} formula f is equivalent to a separated formula.*

Proof Applying Lemma 5 to subformulas $\overline{\langle\langle k\rangle\rangle}\psi$ diminishes the multiset of alternation heights of backward and forward modalities. □

Proposition 3 *(Separation Lemma)*

$$L_{BU} \preceq_s L_{BU}^{sep}$$

is the immediate corollary which combines with

Proposition 4 $L_{BU}^{sep} \preceq_i L_U$.

(same as Proposition 2) to complete the proof of Theorem 3.

Conclusion

Translations between modal logics have not been investigated in the literature. Our three theorems clearly show that many interesting results can be found when modal logics with backward modalities are considered. We intend to pursue this line of research

- by investigating complexity issues (not dealt with in this introductory paper),
- by simplifying our proofs that our rewriting strategies terminate,
- and especially by considering other richer logics: HML with recursion, logics for "truly parallel" models, ...

This last point seems promising. For example, F. Cherief, F. Laroussinie and S. Pinchinat proved that the logic L_P from [2] can be translated into a variant of HML_{bf} with (ρ) modalities for pomsets ρ.

References

[1] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[2] R. De Nicola and G. L. Ferrari. Observational logics and concurrency models. In *Proc. 10th Conf. Found. of Software Technology and Theor. Comp. Sci. Bangalore, India, LNCS 472*, pages 301–315. Springer-Verlag, December 1990.

[3] R. De Nicola, U. Montanari, and F. Vaandrager. Back and forth bisimulations. In *Proc. CONCUR'90, Amsterdam, LNCS 458*, pages 152–165. Springer-Verlag, August 1990.

[4] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. Research Report CS-R9012, CWI, 1990.

[5] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation (extended abstract). In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia, PA*, pages 118–129, June 1990.

[6] D. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Proc. Temporal Logic in Specification, Altrincham, UK, LNCS 398*, pages 409–448. Springer-Verlag, April 1987.

[7] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. Principles of Programming Languages, Las Vegas, Nevada*, pages 163–173, January 1980.

[8] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[9] M. Hennessy and C. Stirling. The power of the future perfect in program logics. *Information and Control*, 67:23–52, 1985.

[10] M. Hillerström. Verification of CCS processes. M.Sc. Thesis, Aalborg University, 1987.

[11] F. Vaandrager. Translating back and forth logic to HML with until operator. Unpublished note, 1992.

Modal Action Logic in a Practical Specification Language

Ismar Neumann Kaufman
ink@di.ufpe.br

Silvio Lemos Meira
srlm@di.ufpe.br

Department of Informatics - Federal University of Pernambuco
P.O. BOX 7851 - 50732-970, Recife-PE, Brazil

1. Introduction

The need for formal specification languages in the requirements phase of software engineering has been recognized by scientists and practitioners alike. The Z language [12], particularly, is widely accepted as a medium to express software requirements, with its schemas providing modularity to build new specifications by composition of elements already defined. Z has been tested in a number of industrial projects.

Nevertheless, schemas are a means of functional decomposition; the last few years have shown that object oriented decomposition is more suitable for the development of large software systems. Object oriented software tends to be more stable through time and enforces extendibility and reusability. Among other ways to bring object orientation to Z [13], **MooZ** [8, 9] was proposed and experimented.

MooZ has many new features, but "respects" Z semantics - based on set theory and first-order predicate calculus - making its application restricted, since properties like temporal ordering of events and concurrency are not easy to describe within such formalisms. The problem gets worse if the language is used for the logical design of software, when we augment the problem universe with elements of the chosen solution. Temporal and concurrency properties appear more often in the solution than in the requirement space.

On the other hand, if we want formalism to permeate software development, we need to extend its application from the requirements phase to latter steps of the software life cycle. A practical approach to formal logical design, based upon the **MooZ** (and Z) experience, capable of treating time and concurrence, among other properties, is the key issue of this work.

The semantic foundation is given by **MAL** [5, 4], a very comprehensive and expressive linguistic framework. In particular, **MAL**'s object structured version is very adequate for the purpose in hand and is described below.

This article reports on the way to incorporate **MAL** in object oriented Z and shows that the approach may be a general way to enrich model based specification languages with stronger semantics.

2. Object structured modal action logic

The application of modal and temporal logics in the specification of software systems has been advocated for more than a decade [1, 2, 11, 10]. The logic shown herein, **MAL**, is adapted from the work of Fiadeiro and Maibaum [5, 4].

A **MAL** specification is a set of related object descriptions, each one being a pair (θ, Φ) where θ is an object signature and Φ is a set of formulas over θ. If an object description is viewed as a theory, the signature and the formulas are the language and the axiomatic of the theory, respectively.

An object signature contains a universal signature (a usual algebraic signature with a special sort for events) and families of attribute and action symbols. The rigid and non-rigid symbols are syntactically distinguished, the former coming from the universal signature and the latter from the attributes and actions. If S is the set of sorts, then every function symbol from the universal algebra and every attribute is $S^* \times S$-indexed; every action is S^*-indexed.

Terms include variables (introduced via classifications), function application (either from universe functions or attributes) and modal qualification of other terms. This last and unusual construction was introduced in [6], in order to make formulas more intuitive, because very often one needs to express the changes in individual entities, not in whole formulas. Languages like VDM and Z have similar features. The translation of our language to **MAL** is easier with modal qualification of terms.

To express change, there are also action terms resulting from the application of action symbols to arguments. Formulas are relations between state propositions.

The semantics of an object signature $\theta = (\Sigma, \alpha, \Gamma)$ is given by an interpretation structure $(\mathcal{U}, \mathcal{J}, \mathcal{P}, \mathcal{O})$, where:

- $\mathcal{U}$ is a Σ-algebra such that $E_{\mathcal{U}}$ (the interpretation of E, the sort of events, in $\mathcal{U}$) is not empty.
- $\mathcal{J}$ maps:
  - $f \in \alpha_{\langle s_1,\ldots,s_n \rangle, s}$ in $\mathcal{J}(f) : s_{1\mathcal{U}} \times \ldots \times s_{n\mathcal{U}} \times E_{\mathcal{U}}^{*} \rightarrow s_{\mathcal{U}}$
  - $g \in \Gamma_{\langle s_1,\ldots,s_n \rangle}$ in $\mathcal{J}(g) : s_{1\mathcal{U}} \times \ldots \times s_{n\mathcal{U}} \times E_{\mathcal{U}}^{*} \rightarrow \wp(E_{\mathcal{U}})$
- $\mathcal{P}$ and $\mathcal{O}$ are relations over $E_{\mathcal{U}} \times E_{\mathcal{U}}^{*}$.

The relations $\mathcal{P}$ and $\mathcal{O}$ state in which event an action is permitted or obligatory. Sequences of actions make trajectories, which can be safe and/or live, following a deontic style of specification that does not *prescribe* behavior. Separating normativeness from inconsistency is richer than the pure temporal logic approach, since it allows the specification of error recovery, punishment, etc. Implementations are either normative or else treat non-normative traces explicitly.

We may define special interpretation structures that guarantee locality. Events that respect locality are called local events. We call ***θ*-locus** an interpretation structure where every non-local event in every trace (finite sequence of events) does not affect any attribute. Locality plays a very important role, assuring that encapsulation of information will be part of the theory presentation. The semantics of modal qualification of terms and state propositions are given over traces, constituting a Kripke semantics, where traces are the "possible worlds".

Satisfaction of propositions is defined in terms of an interpretation structure, an assignment and a trace. The truth-value of formulas is defined by a relation $\rightarrow_{\Xi}$, by which reasoning about information in a state is possible. There is another consequence relation ($\Rightarrow_\theta$) intended to reason about the consequences of a specification: an assertion ($F \Rightarrow_\theta f$) is valid iff every θ-object that makes every formula of F true also makes f true.

Class ⟨*Class_Name*⟩
givensets ⟨*type_names_list*⟩
superclasses ⟨*class_references_list*⟩
⟨*auxiliary_definitions*⟩
private ⟨*definition_names_list*⟩
or
public ⟨*definition_names_list*⟩
constants
⟨*axiomatic_descriptions_list*⟩
⟨*auxiliary_definitions*⟩
state
⟨*anonymous_schema*⟩ or ⟨*constraint*⟩
initialstates
⟨*schema*⟩
⟨*auxiliary_definitions*⟩
operations
⟨*definitions*⟩
EndClass ⟨*Class_Name*⟩.

Figure 1: General structure of a class.

Object descriptions are related to each other by morphisms that map pairs of signatures and axioms. Particularly, morphisms must preserve locality, to allow for compositional development.

3. The new language: MaMooZ

Modular object oriented Z (**MooZ**) [9] enr Z with object-oriented concepts (classes and i itance) keeping the syntax as near as possible to that of Z. Like Z, **MooZ** semantics is based on set theory and classical first-order logic. The language does not allow for definitions outside classes, so that any relation between classes must be either clientship or inheritance. The general format of a **MooZ** class is shown in fig. 1.

MaMooZ[7] is a modal logic enrichment of **MooZ**, where the syntax is close to the latter's but the semantics is given in terms of **MAL**. The translation method is given in [7, Ch.5].

A method may be defined by a schema or by a semantic operation (an axiomatic description which involves state components). The definition of a *method* in a **MaMooZ** class means an action that can be performed by an object in an event. The *events* occur constantly and eternally: there is a *global* event sequence, called *trace*, unique for all the system. Operationally, we can think of events as clock ticks heard by all objects. In some ticks some objects do something, like communicating with other objects or altering their private memory. These actions are specified by the methods.

The methods of an object occur in some subset of the event set. This subset may contain events dispersed throughout the trace. Two methods occurring in the same event are simultaneous; if they come from distinct objects there is a *synchronization* between the objects, maybe with *information exchange*.

Objects can be modified iff one of their methods occur in a given event, otherwise, the event is silent in relation to that object and not observed by it.

In **MooZ**, a method takes in account the object's current state (say s) and the next (s'). This is still valid in **MaMooZ**: the translation of a decorated component is the component modally qualified by the event in which the method occurs, meaning the value of the component after the method's occurrence.

For example, consider the following method definition, specifying a semantic operation that increases the value of a state component. Suppose that a is a component (attribute, in **MAL** terminology) of the class (object description, in **MAL**).

Increases

$a, a' : \mathbf{N}$

$a' > a$

The **MAL** translation of this operation is:

$$\underset{x:E}{\rightarrow} Increases(x) \wedge ([x]a) > a$$

where E stands for the sort of events. This proposition could be read as:

> *When Increases occurs in an event x, the value of a after the event is greater than the value of a before the event.*

The two deontic predicates **Per** and **Obl** are present in **MaMooZ**. **Per**($a_1, \ldots, a_n$) means that some of the methods $a_1, \ldots, a_n$ *may* happen in the next event observed by the object, i.e., that the methods in the list have *permission* to occur.

The predicate **Obl**($a_1, \ldots, a_n$) establishes that in future events the methods $a_1, \ldots, a_n$ will occur. There is no restriction about how many events will fill the trace between the setting of an obligation and its satisfaction. The semantics of an obligation is analogous to that of the operator $\diamond$ (or **F**) in temporal logic: the obligation will be eventually discharged by the occurrence of the method.

Both for **Per** and **Obl** there is no relation between the several methods listed as arguments: they are grouped only for brevity and the order is unimportant. So **Per**($a_1, \ldots, a_n$) is an abbreviation for **Per**(a_1) $\wedge \ldots \wedge$ **Per**(a_n).

Besides **Per** and **Obl**, some few words are introduced in the language to name special sorts. The methods in a class, whether defined or inherited, have sort **Method**. This sort has "local" meaning, its elements being distinct in each context. **Attribute** is the sort of a class' state components and **Event** is the (global) sort of events.

A construct like *object* **Method** could be used to obtain a set with the names of the methods of a class. The same holds for attributes. All these constructs are well founded in **MAL** and, as far as possible, compatible with Z (and **MooZ**) style.

MaMooZ specifications are organized in documents and chapters (coarse grain modules) and classes (fine grain). Operations in the classes may be defined by schemas and axiomatic descriptions. The predicates defining an operation may use deontic predicates (permission and obligation) in order to deal with time and concurrence, but there is no explicit modal qualification of terms, since this is the resource used in the semantics to map components of the operations representing "next state values".

In the full article, we describe the operation of a phone box to show how the resources brought from **MAL** increase the expressive power of the basic specification language.

4. Conclusion

Other approaches to incorporate modal logics in Z are restricted to temporal logics [3]. Richer languages, like **MAL**, may be used too, without many changes to the syntax, with the semantics given by translational approach, instead of ZF theory.

There are many open problems to deal with: the calculus [5] proposed for **MAL** should be "upgraded" to **MaMooZ**, to cater for a more abstract syntactical and semantical discourse.

The adoption of explicit temporal operators should be studied, but care must be taken to avoid conflicts between the deontic and temporal facets. In special, modal qualification of temporal operators is impossible and should be refrained from. Surely, the two tasks are connected: if temporal operators are used, so the calculus must be refined to deal with them.

References

[1] H. Barringer. The use of temporal logic in the compositional specification of concurrent systems. In A. Galton, editor, *Temporal Logics And Their Applications*. Academic Press, 1987.

[2] M. Danelutto and A. Masini. A temporal logic approach to specify and prove properties of finite state concurrent systems. In E. Börger, H.K. Büning, and M.M. Richter, editors, *Proc. CSL'88 2nd Workshop on Computer Science Logic*, Lecture Notes in Computer Science, 1988.

[3] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z Specification Language. Technical Report 91 - 1, SVRC - Software Verification Centre, The University of Queensland, May 1991.

[4] J. Fiadeiro and T. Maibaum. Describing, structuring and implementing objects. In *School/Workshop on Foundations of Object-Oriented Languages, REX/FOOL*, Holanda, 1990.

[5] J. Fiadeiro and T. Maibaum. Towards Object Calculi. In *IS-CORE Workshop*, London, 1990.

[6] J. Fiadeiro and A. Sernadas. Logics of Modal Terms for Systems Specification. *Journal of Logic and Computation*, 1(2):187–227, 1990.

[7] I. N. Kaufman. On the Application of Formal Specification to the Logical Design of Software. Master's thesis, Universidade Federal de Pernambuco, Recife-PE, Brazil, August 1992. In Portuguese.

[8] S. R. L. Meira and A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. In Prof. C. J. van Rijsbergen, editor, *Workshop on Computing Series*, pages 173 - 192, Oxford - Inglaterra, December 1990. Springer-Verlag.

[9] S. R. L. Meira and A. L. C. Cavalcanti. The MooZ Specification Language. Technical report, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1992.

[10] A. Pnueli. The temporal logic of programs. In *Proc. 18th Ann. Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[11] G. Saake and U.W. Lipeck. Using finite-linear temporal logic for specifying database dynamics. In E. Börger, H.K. Büning, and M.M. Richter, editors, *Proc. CSL'88 2nd Workshop on Computer Science Logic*, Lecture Notes in Computer Science, 1988.

[12] J. M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. C. A. R. Hoare, Series Editor. Prentice Hall, 1988.

[13] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.

On using a Composition Principle to Design Parallel Programs
- Extended Abstract -

A. MOKKEDEM and D. MERY*

CRIN-CNRS & INRIA-Lorraine, BP239
54506 Vandœuvre-lès-Nancy Cedex, France
email:{mokkedem@loria.fr,mery@loria.fr}

Abstract

We very briefly present a rigorous and modular method, we are developing to design reactive systems starting from their desired properties. This method is based on a mechanization of Manna-Pnueli's modular validity concept and on a modular temporal language in which properties are invariant under *stuttering* [1]. A compositional proof system is established to support both specification verification and modular program construction. Each program is developed together with the proof that it meets its specification. A refinement relation is defined by using rules in backward, while the proof is constructed by using the same rules in forward. Constrained by a limited space, we hope to focus attention on the underlying concepts and leave a complete presentation of the proof system (soundness, relative completeness, modular completeness, and adaptation completeness) in a future paper. We give some results in this short paper omiting proofs, a full version will include the most results with proofs.

1 Introduction

The temporal logic as presented in [12, 13] provides a powerful tool for *global* specification and *non-compositional* verification of *existing* concurrent programs. However, this logic offers a very poor support for *systematic* design of concurrent programs because of lack of modularity. More recently new concepts have been introduced in order to make the language of temporal logic more *modular* and the temporal proof system more *compositional* [2, 4, 9]. In the present work we explore these new concepts and we present a *modular* specification method together with a *compositional* temporal proof system. We show how our logic offers a rigorous support for the systematic design of concurrent programs.

Our logic aims to provide a mixed verification and development strategy (*top down* and *bottom up*) of concurrent programs. Proof rules should (1) preserve some desired properties (safety and certain liveness properties), (2) be compositional, and (3) be possibly mechanisable. The first feature aims to guarantee that whenever the starting abstract specification expects the system to operate according to some safety properties (partial correctness, deadlock freedom, mutual exclusion, ...) then so behaves the derived implementation. We show that liveness properties are in general more difficult to preserve whenever we want to define the proof rules according to a composition principle. But such a principle is of a great importance when we want to adopt both the *modular verification* and *stepwise refinement* concepts in the concurrency setting. Given the correctness proofs of some small modules, *composition principles* allow the verifier to establish the correctness of bigger modules. Conversely, given the specification of a big module to be implemented, *composition principles* allow the designer to reduce the implementation problem to the subproblems of implementing smaller modules.

Traditionally, composition principles for both specification, verification and refinement of concurrent systems are considered hard to obtain. However, previous work [4, 5, 17] have shown that this difficulty mainly lies in the formulation of a compositional rule for parallel composition. Now, in our opinion, if one wants to formulate a compositional rule for parallel composition, then the first step is to be careful at the stage of the definition of the specification language semantics. Especially, we believe that *invariance under stuttering* of properties[1] is one of the key requirements needed for parallel composition to be conjunction and to be able to implement a coarser-grained program by a finer-grained one in the setting of the temporal logic [2].

2 The logic

The full purpose of this work is to provide a complete methodology for the *compositional* specification, verification and development of reactive programs. For we first introduce a programming notation (IPL) for concurrent modules of a reactive system and define a computational model to represent semantics of modules. The obtained semantics is compositional in the sense that the semantics of a composite reactive systems is computed from a formal relation between semantics of its sub-modules. We then define the temporal logic MTL and derive from it a specification language by establishing a closed connection between computations of IPL program and models of MTL formulas. Our logic is *state-based* oriented. A system may be specified at many level of abstraction; highest-level properties are described in terms of stuttering invariant temporal formulae, while implementations are programs in an intermediate programming language that we call IPL. A highest-level specification must talk about only the expected behaviour of the system, while avoiding references to efficiency or architectural details of its implementation. Such details can be introduced only in the last stage of the *design* process when a parallel algorithmic solution is already built.

*on sabbatical leave at the department of Computing Science University of Stirling under the European Science Exchange Programme Royal Society - CNRS

[1] A property P is said to be invariant under stuttering if whenever a model σ satisfies P then every model τ, stuttering equivalent (this concept will be defined below) to σ satisfies P.

2.1 A programming notation for reactive systems

Reactive systems are coded using the language IPL. This language is a slight modification of the language introduced in [14]. The modifications we have introduced aim to reach a compositional semantics for programs written in IPL. For instance, usual laws, commutativity and associativity, of concatenation and parallel constructs are conserved. The central notion of IPL is the one of *module* statement. Here is an excerpt of the syntax. A module statement has the form M :: [module; *interface*; *body*] where,

$$\begin{array}{l}
\mathit{interface} ::= \{\mathit{modes}\ \mathit{dcl}\}^* \\
\mathit{modes} ::= \{\mathbf{in} |\ \mathbf{out} |\ \mathbf{consum} |\ \mathbf{external}\}^+ \\
\mathit{body} ::= [\mathbf{local}\ \mathit{dcl};]\ \mathit{statement} \\
\mathit{dcl} ::= \{\mathit{variable} \mid \mathit{channel}\}^+ : \mathit{type}\ [\mathbf{where} : \mathit{init}] \\
\mathit{statement} ::= \mathit{action} \mid \mathit{statement}; \mathit{statement} \mid \\
\qquad\qquad \mathbf{IF}\ [\!]_{i=1}^{n} \mathit{guard} \rightarrow \mathit{statement}\ \mathbf{FI} \mid \\
\qquad\qquad \mathbf{DO}\ [\!]_{i=1}^{n}\ \mathit{guard} \rightarrow \mathit{statement}\ \mathbf{OD} \mid \\
\qquad\qquad [\mathit{label}\ :]\mathit{statement}[: \mathit{label}] \\
\mathit{action} ::= \mathit{skip} \mid \mathit{assignment} \mid \mathit{send} \mid \mathit{receive} \\
\mathit{guard} ::= \mathit{expression} \mid \mathit{receive} \\
\mathit{send} ::= \mathit{channel!expression} \\
\mathit{receive} ::= \mathit{channel?variable}
\end{array}$$

A reactive system Net has the following syntax:

$$Net ::= M \mid Net \| Net \mid \nu c.\ Net \mid Net[d/c]$$

Concurrent modules communicate by asynchronous message passing via unbounded channels. Each module should communicate with the environment (other modules) through its interface according to the *modes* assigned to channels. Local variables are not visible outside, thus all variables of a module are implicitly hidden. Throughout the remainder of this paper we assume the syntactic restriction that variables in different modules are distinct, *while we give more attention to channels. Hiding of channels* must be done explicitly using the binder ν. We define the viewed channels of a module M (by the environment) to be channels that are not hidden. $Net[d/c]$ represents *channel renaming* of c into d. Let c be a channel declared in M, a statement of M may have reading (resp. writing) reference to c only if c is declared with the mode in (resp. out). A statement in a module parallel to M may have a reading (resp. writing) reference to c only if c is viewed and declared (in M) with the mode consum (resp. external).

Definition 2.1 (Interface compatibility) *Let M_1 and M_2 be two modules, we say that M_1 and M_2 are interface compatible (we denote by M_1 compat_with M_2) if the declaration for any channel c that is declared as viewed in both M_1 and M_2 satisfy the following requirements: the types of c in both declarations match, the conjunction of the where clauses (supposed true when is not specified) is consistent, and if one of the declarations specifies an out (reps. in) mode, the other specifies an external (resp. consum) mode.*

Semantics. The basic computational model we use to assign semantics to reactive programs is that of fair transition system (*FTS* for short). We associate with each IPL module M a fair transition system $S_M = (\Pi_M, \Sigma_M, \mathcal{T}_M, \Theta_M, \mathcal{J}_M, \mathcal{F}_M)$ which consists of the following components :

Π_M [State variables]: $(= \{\pi_M\} \cup C_M \cup Y_M)$ π_M is a control variable, it ranges over L_M where L_M denotes the set of locations in M. C_M denotes the set of channels declared in the interface of M. Y_M denotes local (*data*) variables in M.

Σ_M [States]: All the possible interpretations of variables in Π_M consistent with their types.

$\mathcal{T}_M$ [Transitions]: these consist of
(1) the transitions τ_S associated with statements S in the body of M, (2) the idling transition τ_I represented by the transition relation ρ_{τ_I} : *true*; it represents the *stuttering* steps in Abadi-Lamport's terminology [1] which characterise internal transitions executed by the environment, (3) the *environment receiving* transition τ_b^{ER}, i.e. the transitions specified by the formula $(|b| > 0) \wedge (b' = tl(b))$ for any *consum* channel $b \in C_M$ (4) the *environment sending* transition τ_c^{ES}, i.e. the transitions specified by the formula $\exists u.\ (c' = c \bullet u)$ for any *external* channel $c \in C_M$. τ^{ES} and τ^{ER} represent the observable transitions executed by the environment. We denote, for a module M, the environment transitions by $\mathcal{T}_M^e$ and by $\mathcal{T}_M^i$ all the other ones.

Θ_M [Initial condition]: consists of $\Theta_M = (\pi = l_0) \wedge \varphi$, where φ represents the *where* parts of the declarations of out channels and local variables (l_0 is the initial location of the module M). The initial value of external channels is controled by the environment.

Fairness: $\mathcal{J}_M$ contains just transitions, i.e. transitions which cannot be continually enabled but tacken only finitely many times. This consists of all the (internal) transitions associated with local statements of M. $\mathcal{F}_M$ contains fair transitions i.e. transitions which cannot be infinitely often enabled but taken only finitely many times. This consists of all the (observable) transitions associated with communication statements of M. Environment transitions such as τ_I, τ^{ER}, and τ^{ES} are contained in neither $\mathcal{J}_M$ nor $\mathcal{F}_M$.

Behaviours: A behaviour of a module M is a set of computation structures which represent its possible executions. A (possible) computation *of M is an infinite sequence of* states $\sigma : s_0, s_1, \ldots$ such that (1) s_0 satisfies the initial condition Θ_M, (2) for each $i \geq 0, \tau(s_i, s_{i+1})$ for some $\tau \in \mathcal{T}_M$, (3) σ satisfies *justice* and *fairness* requirements imposed by the sets $\mathcal{J}_M$ and $\mathcal{F}_M$. Two computations σ, τ are said to be stuttering equivalent (in notation $\sigma \simeq \tau$) if they are equal modulo stuttering steps. We recall that in such a semantic model, finite computations are represented by infinite sequences by adding an infinite number of stuttering steps (τ_I) which takes the halting state into itself.

The semantics of a reactive system $N_1 \| \ldots \| N_n$ is a *fair* transition system resulting from a *fair* composition of transition systems associated with modules[2] N_i, in notation, $S_{N_1\|\ldots\|N_n} = S_{N_1} \otimes \ldots \otimes S_{N_n}$. Executions in $S_{N_1\|\ldots\|N_n}$ are represented as *interleaving* concurrent actions in the different modules under *fairness* constraints.

Definition 2.2 *Let M_1 and M_2 be two channel-hiding free modules (i.e., modules in which the binder ν does not occur). We define $int\_mode_i(c)$ (resp. $aux\_mode_i(c)$) to be the set of modes in $\{\underline{in}, \underline{out}\}$ (resp. in $\{\underline{consum}, \underline{external}\}$) assigned to the channel c in the module M_i, and $mode_i(c) = int\_mode_i(c) \cup aux\_mode_i(c)$. We denote by $\overline{aux}\_mode(c)$, for a shared channel c, the set of modes m such that $m \in (aux\_mode_1(c) \cup aux\_mode_2(c)) \setminus (aux\_mode_1(c) \cap aux\_mode_2(c))$.*

Definition 2.3
Let $S_{M_i} = (\Pi_{M_i}, \Sigma_{M_i}, \mathcal{T}_{M_i}, \Theta_{M_i}, \mathcal{J}_{M_i}, \mathcal{F}_{M_i})$, $i \in \{1, 2\}$ be the FTS associated with modules M_1 and M_2. The FTS associated with the composed module $M_1 \| M_2$ is defined as

[2] Although components N_i are arbitrary reactive systems we call them modules; semantically we consider a reactive system as a new composed module.

follows:
$S_{M_1 \| M_2} = (\Pi_M, \Sigma_M, \mathcal{T}_M, \Theta_M, \mathcal{J}_M, \mathcal{F}_M)$ *such that*

1. $\Pi_M = \Pi_{M_1} \cup \Pi_{M_2}$ *where, (i) for every non-shared channel c (i.e. c is declared in M_j only for $i \in \{1,2\}$), $mode(c) = mode_j(c)$, and (ii) for every shared channel c (i.e. $c \in \Pi_{M_1} \cap \Pi_{M_2}$), $int\_mode(c) = int\_mode_1(c) \cup int\_mode_2(c)$ and $aux\_mode(c) = aux\_mode_1(c) \cap aux\_mode_2(c)$.*

2. $\Sigma_M = \{s : \Pi_M \rightarrow D_M / s|_{\Pi_{M_1}} \in \Sigma_{M_1}$ *and* $s|_{\Pi_{M_2}} \in \Sigma_{M_2}\}$

3. $\mathcal{T}_M = (\mathcal{T}_{M_1} \cup \mathcal{T}_{M_2}) \setminus (\{\tau_C^{ES} / c \in \Pi_{M_1} \cap \Pi_{M_2} \wedge \underline{external} \in \overline{aux}\_mode(c)\} \cup \{\tau_C^{ER} / c \in \Pi_{M_1} \cap \Pi_{M_2} \wedge \underline{consum} \in \overline{aux}\_mode(c)\})$

4. $\Theta_M = \Theta_{M_1} \wedge \Theta_{M_2}$ *(consistency is guaranteed by the interface compatibility requirement)*

5. $\mathcal{J}_M = \mathcal{J}_{M_1} \cup \mathcal{J}_{M_2}$ *and* $\mathcal{F}_M = \mathcal{F}_{M_1} \cup \mathcal{F}_{M_2}$

We complete the definition given above by the following laws to deal with hiding and renaming of channels.
$M_1 \| \nu c.M_2 \approx^3 \nu c.(M_1 \| M_2)$ if $c \notin chan(M_1)$
$M_1 \| \nu c.M_2 \approx \nu d.(M_1 \| M_2[d/c])$ if $c \in chan(M_1)$, where d is a fresh channel variable.
Now let M an arbitrary module whose the associated *FTS* is $(\Pi_M, \Sigma_M, \mathcal{T}_M, \Theta_M, \mathcal{J}_M, \mathcal{F}_M)$,
(1) $S_{\nu c.\ M} = (\Pi_M, \Sigma_M, \mathcal{T}_M, \Theta'_M, \mathcal{J}_M, \mathcal{F}_M)$ with $\Theta'_M = \Theta_M \setminus \{\tau_c^{ER}, \tau_c^{ES}\}$.
(2) $S_{M[d/c]} = S_M[d/c]$ (renaming is extended to tuples in the usual way).

Definition 2.4 (compatible computations) *Let σ, τ be two computations of S_{M_1} and S_{M_2} respectively (M_1 and M_2 are expected to be executed in parallel, so they are interface compatible), we say that σ and τ are compatible ($\sigma \bowtie \tau$) iff all transitions in σ and τ involving shared channels are observable by each other (a formal definition is given in the full paper).*

Proposition 2.1 *The relation compat_with satisfies the following properties:*
(i) Let M_1 and M_2 be two modules, if M_1 compat_with M_2 then M_2 compat_with M_1,
(ii) Let M_1 and M_2 be two compatible modules. M compat_with $[M_1 \| M_2]$ iff M compat_with M_1 and M compat_with M_2

Proposition 2.2 *Let $M = [M_1 \| M_2]$ and S_M the FTS associated with M according to the relation $S_M = S_{M_1} \otimes S_{M_2}$, the two following propositions are equivalent:*
(1) *there exists a computation of S_M such that $\sigma|_{\Pi_1} \simeq \sigma_1$ and $\sigma|_{\Pi_2} \simeq \sigma_2$*
(2) *σ_1 and σ_2 are two compatible computations of S_{M_1} and S_{M_2} respectively.*

Another important consequence of the definition 2.3 is the reaching of the usual laws of parallel construction.

Proposition 2.3 *Let M_1, M_2, M_3 be three interface compatible modules,*
$M_1 \| M_2 \approx M_2 \| M_1$
$(M_1 \| M_2) \| M_3 \approx M_1 \| (M_2 \| M_3)$

2.2 A stuttering invariant temporal language

We are convinced that we must be careful at the design decision stage when we want to define a temporal logic for reactive programs which should be compositional. Linear discrete temporal logic has been perceived to be an appropriate tool for both description of semantics of concurrent (and sequential) programs and reasoning about them. This relies on the fact that concurrent program behaviour can be easily modelled by all possible interleavings of the discrete, linear, execution sequences arising from the separate 'sequential' processes of the concurrent programs (interleaving semantics). In [3] Barringer *et al.* proposed a compositional temporal logic for the specification and verification of concurrent systems. They use a *floating* version of the linear temporal logic with the fixpoint operators and still represent actions by the classical *Next* operator ○. However, such a logic has been strongly criticized from different points of view. Our study of a refined temporal logic, namely MTL, starts from a list of valid claims made by the pioneers of the temporal logic:

- In [7] Lamport objects the use of the *Next* operator to be the origin of some trouble in abstraction, which forces too much irrelevant detail to be present in the semantic description. It appears that the lowest level of atomicity is forced to be visible, which a properly abstract semantics should not make. He provided a strong evidence that all the properties one wishes to express for asynchronous systems do not require this operator.

- Still for abstraction, quantification over state variables turned out to be very useful [11], and has been shown to be necessary for attaining compositional completeness.

- Manna and Pnueli [10] argued the addition of the *past* fragment to the *future* temporal logic to contribute to the utility of the temporal language; while it is not more expressive the full language is found to be more convenient. In [11] they gave some points of dissatisfaction of the full logic presented in [10] due to the *floating* interpretation which does not assign any special significance to the initial state so that satisfiability and validity are evaluated at *all* positions in the computations. This interpretation needs the generalization rule in the proof system which violates the *deduction* rule (a powerful tool in the predicate calculus) and, in the other hand, requires the *suffix* closure property for the set of computations when one needs to interprete formulas over computations of a given program. In fact, they presented an *anchored* temporal logic in [11] in which they consider that a formula φ is defined to be *valid* (resp. *satisfiable*) over a set of sequences C, if it holds at position 0 of *every* (resp. some) sequence of C.

Our present contribution is concerned with the investigation of such remarks and the proposition of a refined *future*-temporal logic MTL in which (1) we consider the *anchored* interpretation, (2) quantification over state variables, and (3) actions are formulated in terms of a new *Next* operator which is insensitive to finite stuttering and sensitivity to infinite stuttering. The resulting logic has the same expressibility power than the full temporal logic [14], does not require *suffix* closure of program computations, and guarantees *invariance under stuttering* of properties. It thus provides a good abstraction for compositional specification and verification of concurrent systems and also offers a good support for systematic design of concurrent programs.

The new and central concept in the definition of MTL consists in introducing a new kind of *Next* operator, denoted

[3] Two programs N_1 and N_2 are semantically equivalent if their associated *FTSs* S_{N_1} and S_{N_2} are equivalent; what we write $N_1 \approx N_2$.

$\otimes_w$, (and its dual, denoted $\oplus_w$). An important feature of $\otimes_w$ is being *insensitive* to finite w-stuttering and *sensitive* to infinite w-stuttering (with respect to a given set of variables w), while its dual, $\oplus_w$, is *insensitive* to both finite and infinite w-stuttering. We then define the other temporal operators (*until*, *unless*, etc.) in terms of $\otimes_w$ in order to obtain a temporal logic that will enable semantic descriptions which are *invariant* under finite w-stuttering. This is one of the major results to ensure a *desired* level of abstraction necessary for modular specification and compositional verification of concurrent systems.

2.3 *Properties of IPL programs*

In order to relate a specification presented by a formula in the logic to the program it is supposed to specify, it is necessary that the computations of a program can serve as *models* (in the logical sense) for the formula, which means that we can evaluate the formula on each of these computations and find whether it holds on the computation. Then, we say that the program satisfies (or implements) the specification given by the formula φ if φ holds over each of the computation of its behaviour. For we augment the MTL logic by some program specific predicates and functions, referring to the additional IPL constructs needed to fully describe a state in the computation of a reactive program, for instance, the *control*-predicates like *at(M)*, *after(M)*, and the *mode*-predicates like $out(c), c\text{-}out(c), in(c), \ldots$.

One of the most important classifications of properties of reactive systems, is their partition into safety and liveness properties [8]. The advantage of this partition is to provide a way to recognize some incompleteness aspects of specifications. For example, it is now well known that no *specification of a system can be complete without containing* some *safety* properties and some *liveness* properties. In most cases all the safety properties can be trivially satisfied by a program that does nothing. We may view one of the roles of liveness properties as ensuring that safety properties are not implemented by a "do-nothing" program. They are hence intented to discard trivial solution during the design process. A property of an IPL program is of the form:

$$\begin{array}{l}
p \longrightarrow_w q =_{df} \Box_w(p \Rightarrow \otimes_w q)\\
p\ \underline{unless}\ _w\ q =_{df} p \wedge \neg q \longrightarrow_w p \vee q\\
\mathrm{stable}_w(p) =_{df} p\ \underline{unless}\ _w\ \mathrm{false}\\
invariant_w(p) =_{df} p \wedge \mathrm{stable}_w(p)\\
p \leadsto_w q =_{df} \Box_w(p \Rightarrow \Diamond_w q)
\end{array}$$

2.4 Modular specification

Large systems are built up of several components (modules) and a separate specification is given for each component specifying its desired behaviour in the whole system. For specifying concurrent modules we explore Lamport's modular specification method [6, 8] and similar notions introduced in [14]. We should be able to separately specify concurrent program modules in a convenient way as in work of Lamport [6, 8]. We emphasize, in particular, the relevance to complement a specification module by the specification of the *interface*–the mechanism by which the module communicates with its environment. The interface specification of a module stipulates the constrains the environment must satisfy for a correct communication with this module. Thus a specification module consists of two parts: The first part, namely *interface*, specifies constraints on the interaction of the system with its environment. The second part, namely *body*, specifies the computation expected from the system. It specifies the initial state of the system, its safety properties and its liveness properties. The information that should contain the interface is especially essential for the completeness of the specification module. For this purpose, Lamport argued in [8] that the interface must be specified at the implementation level. Indeed a complete specification should eliminate the need for any communication between the user of the module and its implemenator. Thereby the *interface* part will be a low-level specification (IPL), while the *body* a highest-level specification (temporal language).

Definition 2.5 (Module specification) *A module specification is an object of the form* [inter; var; φ] *where inter declares shared variables (channels) and var declares local variables. φ is a MTL formula which specifies the expected behaviour of the module within the whole system. The property φ has to be satisfied independently of the context in which the module operates.*

Definition 2.6 (Modular validity [14]) *A formula φ is defined to be modularly valid for a module M_1 (in notation, $M_1 \supset\!= \varphi$) if φ is valid over the program $M_1||M_2$ for any module M_2, interface compatible with M_1 (in notation, $M_1||M_2 \models \varphi$).*

Lemma 2.1 *If $M_1 \supset\!= \varphi_1$ and M_2 compat_with M_1 then* $[M_1||M_2] \supset\!= \varphi_1$.

Overview. The proof system provides a collection of composition proof rules which, on the one hand, given the correctness proofs of some small modules, allow the verifier to establish the correctness of bigger modules. Conversely, given the specification of a big module to be implemented, allow the designer to decompose (or refine) the big specification into less abstact (or more *strong* in the logical sense) ones whose implementations could be found more easy. More precisely, from a design point of view, starting from the specification of the reactive system, the method will assist the designer to refine it into more elementary (but modular) ones. This arises to specifications such that when we put their implementations together (in parallel) we will obtain a parallel implementation for the first specification. Refinement is carried out together with a proof methodology. Once a refinement step is done, one can make sure that it preserves the set of solutions of the first specification. There are two kinds of proofs we have to make during a refinement: a consistency proof which checks whether a specification is mathematically consistent, and a refinement proof which verifies a refinement to be consistent with the specification from which it is refined. The elementary specifications can be refined again according the same principle. This refinement process proceeds until implementations (modules in the IPL) should be obviously derived. Now program texts are difficult to derive directly from weak eventuality so that no program can be extracted while the specification contains weak eventuality. We first transform weak eventuality to strong eventuality preserving invariants. Axioms in the proof system represent the basic laws of the refinement which (in our study) allow to derive atomic actions in a module from its local strong eventuality properties. The rules for composition and refinement of complex specifications are formulated as sound inference rules.

3 A small example

We look for a program $?M$ (maybe a parallel one) that implements the specification module $[Inter; loc; \exists_w \tilde{z}.\ I \wedge$

$S \wedge L]$ where, $w = \{c, u, v\}$, and

$$\begin{array}{ll}
Inter \equiv \underline{\text{c-out}}(c) \wedge c \in \mathbb{N}^* & \text{(interface state functions)} \\
loc \equiv u \in \mathbb{N} \wedge v \in \mathbb{N} & \text{(local state functions)} \\
I \equiv at(M) \wedge u = 0 \wedge v = 0 \wedge c = \epsilon & \text{(initial condition)} \\
S \equiv u = 0 \ \underline{unless}\ _w u = 1 \wedge v = 0 \ \underline{unless}\ _w v = 1 \wedge & \\
\qquad c = \epsilon \ \underline{unless}\ _w c = 1 \bullet \epsilon \wedge after(M) & \text{(safety)} \\
L \equiv at(M) \leadsto_w after(M) & \text{(liveness)}
\end{array}$$

So the starting goal is $?M \supset\!= [Inter; loc; \exists_w \vec{z}.\ I \wedge S \wedge L]$.

If we apply the composition rule we must find sub-formulae which satisfy the proof obligations associated with this rule.

Let $\varphi_w =_{df} I \wedge S \wedge L$ and $\varphi_i =_{df} I_i \wedge S_i \wedge L_i$,

$$\frac{\begin{array}{l} M_1 \supset\!= [inter_1; loc_1; \varphi_1] \\ M_2 \supset\!= [inter_2; loc_2; \varphi_2] \end{array}}{[M_1 \| M_2] \supset\!= [Inter; loc; \varphi]} \quad \left\{ \begin{array}{l} \text{Proof obligations :} \\ \varphi_1 \wedge \varphi_2 \Rightarrow \varphi \\ inter_1 \ \textbf{compat\_with}\ inter_2 \\ Inter = inter_1 \oplus inter_2 \\ loc = loc_1 \oplus loc_2 \\ Var(loc_1) \cap Var(loc_2) = \emptyset \end{array} \right.$$

$$\frac{M \supset\!= [inter; loc; \varphi_w]}{\nu \vec{z}.\ M \supset\!= [Inter; loc; \exists_w \vec{z}.\ \varphi_w]} \quad \left\{ \ Inter = inter \ominus \vec{z} \right.$$

A possible solution is : (where $w_1 = \{c, c_1, c_2, u\}$)

$$\begin{array}{l}
inter_1 \equiv \underline{\text{c-out}}(c, c_1) \wedge \underline{\text{ext-in}}(c_2) \wedge c \in \mathbb{N}^* \wedge c_1 \in \mathbb{N}^* \wedge \ c_2 \in \mathbb{N}^* \\
loc_1 \equiv u \in \mathbb{N} \\
I_1 \equiv at(M_1) \wedge u = 0 \wedge c = \epsilon \wedge c_1 = \epsilon \\
S_1 \equiv (u = 0 \wedge c = \epsilon \wedge c_1 = \epsilon)\ \underline{unless}\ _{w_1}(u = 0 \wedge c = \epsilon \wedge c_1 = 1 \bullet \epsilon) \\
\wedge(u = 0 \wedge c = \epsilon \wedge c_2 = 1 \bullet \epsilon)\ \underline{unless}\ _{w_1}(u = 1 \wedge c = \epsilon \wedge c_2 = \epsilon) \\
\wedge(u = 1 \wedge c = \epsilon)\ \underline{unless}\ _{w_1}(u = 1 \wedge c = 1 \bullet \epsilon \wedge after(M_1)) \\
L_1 \equiv at(M_1) \leadsto_{w_1} after(M_1)
\end{array}$$

with a symmetrical solution for φ_2 (with a slight difference that c does not appear in φ_2) where $\{u, c_1, c_2\}$ in φ_1 are replaced by $\{v, c_2, c_1\}$ and w_1 is replaced by w_2 which is equal to $\{c_1, c_2, v\}$.

We can now apply another rules and/or axioms to find two possible modules $M1$ and $M2$ which must satisfy the two new specification modules. This leads to a final (possible) solution $?M = \nu c_1, c_2.\ [M_1 \| M_2]$ with

$$M1 :: \left[\begin{array}{l}
\underline{\textbf{module}} \\
\textbf{external in } c_2 \ : \textbf{channel } [1..] \textbf{ of integer} \\
\textbf{consum out } c_1,\ c \ : \textbf{channel } [1..] \textbf{ of integer} \\
\textbf{local } u \ : \textbf{integer where } u = 0 \\
\quad \left[\begin{array}{l} l_0 : c_1!1; \\ l_1 : c_2?u; \\ l_2 : c!1 \end{array}\right]
\end{array}\right]$$

and

$$M2 :: \left[\begin{array}{l}
\underline{\textbf{module}} \\
\textbf{external in } c_1 \ : \textbf{channel } [1..] \textbf{ of integer} \\
\textbf{consum out } c_2 \ : \textbf{channel } [1..] \textbf{ of integer} \\
\textbf{local } v \ : \textbf{integer where } v = 0 \\
\quad \left[\begin{array}{l} m_0 : c_1?v; \\ m_1 : c_2!1 \end{array}\right]
\end{array}\right]$$

<u>Program Ping-Pong</u>

4 Discussion

Our previous work has concerned the verification of concurrent programs using a linear temporal logic in which we considered the global validity notion [15, 16]. This led to a non-compositional proof system we have encoded into Isabelle [18]. The resulting prototype, called CROCOS, allows concurrent programs to be verified but not to be developed. It also suffers from inefficiency when dealing with programs of a realistic size.

We intend to improve the current version of CROCOS by implementing the principles and the logic we have reported in this paper. This should permit our prototype to support the verification as well as the derivation of large reactive systems. Then experiments and case studies with the new CROCOS will be conducted in order to explore the possibility of defining other composition principles in the setting of our logic.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Third Annual Symposium on Logic In Computer Science*, pages 165–177, Edinburgh, July 1988.

[2] M. Abadi and L. Lamport. Composing specifications. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Springer Verlag, 1990. LNCS 430.

[3] H. Barringer. The use of temporal logic in the compositional specification of concurrent systems. In A. Galton, editor, *Temporal logics and their applications*, pages 53–90, London, 1987. Academic Press.

[4] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Sixteenth ACM Symposium on Theory of Computing*, pages 51–63, April 1984. ACM.

[5] L. Lamport. The 'Hoare Logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[6] L. Lamport. Specifying concurrent program modules. *ACM Transactions On Programming Languages And Systems*, 2(5):190–222, april 1983.

[7] L. Lamport. What good is temporal logic? pages 657–677. IFIP, 1983.

[8] L. Lamport. A simple approach to specifying concurrent systems. *Communications of ACM*, 1(32):32–45, January 1989.

[9] L. Lamport. The temporal logic of actions. Technical report, DEC Palo Alto, December 1991.

[10] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, pages 196–218. Spinger Verlag, 1985. LNCS 193.

[11] Z. Manna and A. Pnueli. The anchored version of teh temporal framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 201–284, New York, 1981. Spinger Verlag. LNCS 354.

[12] Z. Manna and A. Pnueli. Verification of concurrent programs: A temporal proof system. In *4th School on Advanced Programming*, pages 163–255, June 1982.

[13] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *Correctness Problem in Computer science*, pages 215–273, London, 1982. Academic Press.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991. ISBN 0-387-97664-7.

[15] D. Méry and A. Mokkedem. A proof environment for a subset of SDL. In O. Faergemand and R. Reed, editors, *Fifth SDL Forum Evolving methods*. North-Holland, 1991.

[16] D. Méry and A. Mokkedem. Crocos: An integrated environment for interactive verification of SDL specifications. In G. Bochmann, editor, *Computer-Aided Verification Proceedings*. Springer Verlag, 1992.

[17] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[18] L. Paulson and T. Nipkow. Isabelle tutorial and users's manual. Technical report, University of Cambridge, Computer Laboratory, 1990.

A notion of refinement for automata

N. Sabadini, S. Vigna

Dipartimento di Scienze dell'Informazione,

Università di Milano, Via Comelico 39/41, I-20135 Milano MI, Italy

FAX: +39-2-55006276; e_mail: sabadini@imiucca.csi.unimi.it, vigna@ghost.dsi.unimi.it

R.F.C. Walters

School of Mathematics and Statistics, University of Sydney, N.S.W. 2006, Australia

e_mail: walters_b@maths.su.oz.au

1. Introduction

In this paper we discuss a notion of morphism of automata which seems particularly appropriate for the study of concurrency and distributed processing. It has close connections with notions of morphism introduced in [MM90] and [Knu73].

We are following the automata-theoretic calculus for concurrency based on distributive categories introduced by Sabadini and Walters in [SW93], which we recall briefly below.

We concentrate attention here in particular on refinement. Our notion differs markedly from existing notions of refinement in process calculi [DGR], and Petri nets [BGV91], which were introduced with a view to top-down design; hence, in these approaches it is considered desirable that that the behaviour and the properties of a refined system are deducible from the unrefined one. For example, two equivalent systems (with respect to a given notion of equivalence) must be equivalent even after refinement. As a consequence, no new information is being introduced by a refinement. On the contrary, in our approach, the refined system may have a much richer structure than the unrefined one: thus, we can study such issues as efficiency in time and resources. This implies that at each stage of refinement it is necessary to prove that desired properties are preserved by refinement. This is not unreasonable, because the desired properties are properties of the *final* object of these refinement process, and at an earlier stage it may happen that it is not even possible to define them. This approach is advocated, for instance, by Chandy and Misra in [CM88].

Formally, our definition is based on considering the automata as categories of transitions, and then a morphism is a functor between transition categories, and a refinement is an embedding of the category of one automaton in another one. The elementary categorical concepts used in this paper may be found in Mac Lane [Mac71] or Walters [Wal].

In future papers we will show how the morphisms introduced here can be used to prove properties of distributed systems.

This work has been supported by the Australian Research Council, Esprit BRA ASMICS, Italian MURST 40%, and the Italian CNR, contract 92.00529.CT01.

2. Distributive Automata

In the model of concurrency introduced in [SW93], sytems are represented by particular deterministic automata called distributive automata.

Distributive automata are automata constructed from a given family of sets and function (data types and data operations) using the operations of a distributive category. That is, the alphabet and state space of a distributive automaton is formed by the operations of sum and product from some basic sets. The actions of a distributive automaton are formed from basic functions by composition, sum, and product of functions, projections, injections, the diagonal and codiagonal, and the distributive isomorphism $X \times (Y + Z) \cong X \times Y + X \times Z$. Thus, the alphabets have a rich structure reflecting parallel or conflicting, synchronous or asynchronous actions.

There is one further operation. A distibutive automaton whose alphabet is one letter and whose state space is of the form $X + U + Y$ may compute by iteration a (total) function from X to Y; such automata we call pseudofunctions. In the construction distributive automata we may use the function computed by a pseudofunction. This operation allows hiding of state, and encapsulation of iteration. Notice that the notion of pseudofunction has a precursor in Elgot's iteration theories [Elg75] and Heller's work on recursion categories [Hel90]. A similar definition can also be found in [Knu73], and [Mil71].

3. Refinement of automata

Definition 3.1. Suppose M is a monoid and $\mathbf{X}$ an M-automaton; that is, a set X together with an action of M on X, $M \times X \rightarrow X : (m, x) \mapsto m \cdot x$; the action is required to satisfy the usual axioms $m_1 \cdot (m_2 \cdot x) = (m_1 m_2) \cdot x$ and $1 \cdot x = x$. Define the category *Trans*($\mathbf{X}$) (the *transition category of* X) as follows:

(i) objects are states (that is, elements) of X;
(ii) arrows from x to y are state transitions; that is, elements $m \in M$ such that $m \cdot x = y$;
(iii) composition is monoid multiplication.

A morphism of automata, or, in short, a *mapping* from $\mathbf{X}$ to $\mathbf{Y}$, is a functor from *Trans*($\mathbf{X}$) to *Trans*($\mathbf{Y}$), where $\mathbf{Y}$ is an N-automaton for a monoid N.

An *abstraction* from $\mathbf{X}$ to $\mathbf{Y}$ is a functor *Trans*($\mathbf{X}$) to *Trans*($\mathbf{Y}$) which is surjective on objects and arrows.

A *refinement* of $\mathbf{X}$ in $\mathbf{Y}$ is an inclusion, as a full subcategory, of *Trans*($\mathbf{X}$) in *Trans*($\mathbf{Y}$).

In other words, in order to give a refinement of $\mathbf{X}$ one has to specify a bigger system $\mathbf{Y}$ which has a restriction to a system isomorphic to $\mathbf{X}$.

Notice that each arrow in *Trans*($\mathbf{X}$) is determined by an element $m \in M$ *and* a domain and a codomain $x, y \in X$. Hence many distinct arrows will be labelled with the same element of the monoid.

In what follows, we will be concerned with free monoids on the structured alphabets we discussed. If $M = A^*$ and $N = B^*$, a functor F from *Trans*($\mathbf{X}$) to *Trans*($\mathbf{Y}$) is given by a function $F : X \rightarrow Y$ and a function $F : X \times A \rightarrow B^*$ satisfying the condition that if $\alpha \in A$ and $\alpha : x \rightarrow x'$ in *Trans*($\mathbf{X}$), then $F(\alpha) : F(x) \rightarrow F(y)$ in *Trans*($\mathbf{Y}$). For a refinement there is the further requirement that the function induced by F between $\mathbf{Hom}(x, x')$ and $\mathbf{Hom}(F(x), F(x'))$ is a bijection, and that the function between the state spaces in injective. (Morphisms of distributive automata should be defined by functions $X \rightarrow Y$ constructed using the operations of a distributive category, and by functions $A \times X \rightarrow B^*$ constructed using the operations of a countably extensive category with products [KWW], but this requirement is not necessary for the purposes of the present paper.)

Notice that the usual notion of substitution in language theory is a morphism which assigns to a letter a word or a language, but the latter ones are fixed once for all, and not dependent on state. Note also that not all full subcategories of $\mathbf{Y}$ induce a refinement.

We can give also the following, weaker

Definition 3.2. An *expansive mapping* is an inclusion F of *Trans*($\mathbf{X}$) in *Trans*($\mathbf{Y}$) such that whenever $F(x \xrightarrow{\alpha} x') = F(x) \xrightarrow{s} F(x')$, where $\alpha \in A$ and $s \in B^*$, then there are no $x'' \in X$, $s' \in B^*$ such that s' is a proper prefix of s and $F(x) \xrightarrow{s'} F(x'')$

When an atomic action is refined by an expansive mapping, the set of states spawned by the string it is mapped to lies entirely outside of the image of **X**, except for the initial and final states (which are the image of the domain and of the codomain of the atomic action). We can indeed restate Definition 3.2 as follows:

(i) $Y = X + U$ for some set U;
(ii) if $F(x, a) = b_1 \cdots b_n \in B^*$, then $b_1 \cdots b_k \cdot x \in U$ for $k = 1, 2, \ldots, n-1$.

Expansiveness and fullness are related by the following proposition:

Proposition 3.1. Let **X** and **Y** be A^* and B^* automata, respectively. If a mapping $F : \mathbf{X} \to \mathbf{Y}$ is a refinement, then it is expansive.

Proof. Suppose there are x'' and s' as in Definition 3.2. Then s factors as $s's''$, and $F(x'') \xrightarrow{s''} F(x')$. But because of faithfulness and fullness, there has to exist strings $t', t'' \in A^*$ such that $x \xrightarrow{t'} x''$ and $x'' \xrightarrow{t''} x'$. By composition, we get $t't'' = \alpha$. Thus, either $t' = \alpha$ and $t'' = \epsilon$, or $t' = \epsilon$ and $t'' = \alpha$. In both cases, s' is not a proper prefix of s. □

This proposition cannot be reversed. Take $A = B = \{\alpha, \beta\}$, $X = \{*\}$ and $Y = \{0, 1\}$. Let the action of α be the identity on Y, and the action of β be $n \mapsto 1 - n$. The mapping sending the unique state of X into 0, α to α and β to $\beta\beta$ is expansive, but not full.

There is however a relevant case in which we can reverse Proposition 3.1:

Proposition 3.2. Let **X** and **Y** be A^* and B^* automata, with $A = B = \{\tau\}$. If a mapping $F : \mathbf{X} \to \mathbf{Y}$ is expansive, then it is a refinement.

Proof. If F is not a refinement, consider states $x, y \in X$ and an arrow $F(x) \xrightarrow{\tau^k} F(y)$ which is not image of an arrow from x to y. Assume without loss of generality that k is minimal. Let $F(x) \xrightarrow{\tau^n} F(z)$ be the image through F of $x \xrightarrow{\tau} z$. If $k > n$, then necessarily $F(z) \xrightarrow{\tau^{k-n}} F(y)$ is not in the image of **X**, which contradicts the minimality of k. Then $n > k$. But this contradicts expansiveness. □

Example 3.1. When $M = N = \{\tau\}^*$ then refinement takes a particularly simple form. Such an automaton can be analyzed by considering the *orbits*, that is, sequences of states produced by the action of τ starting from a given initial state. A *refinement* of an automaton **X** is another automaton **Y** with state space of the form $Y = X + U$ such that the orbits of **Y**, when restricted to **X**, correspond exactly to orbits of **X**.

Remark 3.1. It is clear that refinements form a category **Refine**, and that abstractions form a category **Abstract**. However, both refinement and abstraction can be looked at in the opposite direction, i.e., the domain of a refinement can be seen as a system in which space and time have been hidden, while the domain of an abstraction can be seen as a system with finer state space and actions (this is closely related to [Lyn87]). Formally, this correspond to the study of the categories $\mathbf{Refine}^{op}$ and $\mathbf{Abstract}^{op}$.

The notion of transition category induces a notion of *behaviour* which is state dependent: for each pair of states x, y we can build the set of arrows between x and y, i.e., the *hom-set* between the objects x and y. Formally,

Definition 3.3. The functor behaviour

$$\mathbf{Behaviour} : \mathbf{Refine} \to \mathbf{Cat/Sets}$$

is defined by

$$\mathbf{X} \mapsto \mathbf{Hom} : \mathit{Trans}(\mathbf{X})^{op} \times \mathit{Trans}(\mathbf{X}) \to \mathbf{Sets}$$

on objects, and by

$$F \mapsto F^{op} \times F$$

on morphisms.

Note that $F^{op} \times F$ commutes with **Hom** up to isomorphisms exactly because F is a refinement. Note also that $F^{op} \times F$ is a morphism in **Cat/Sets**; this expresses the fact that the behaviour of $\mathbf{X}$ is a restriction of the behaviour of $\mathbf{Y}$ along the refinement.

4. Examples

4.1. *Mutual exclusion*

Other theories of refinements often require that all the steps in the refinements of two conflicting action (systems) are conflicting. This seems to be reasonable when the word "conflict" means "irrevocable choice", but not when, as usual in applications, conflict comes from access to a common resource (in our setting, this means that two letters use the same part of the state space). Here, we can easily model the situation where the conflict may occur at only one step in the refinement.

4.2. *Independent actions are not necessarily parallel*

In considering a refinement $F : \mathbf{X} \to \mathbf{Y}$ we can think of $\mathbf{X}$ as the specification of a program and F as the implementation of $\mathbf{X}$ in a system $\mathbf{Y}$ (in a later paper we will discuss a more general notion of specification in this setting). It is then possible to consider questions of resources. We can make the distinction between actions of $\mathbf{X}$ being "independent" and being "parallel". Actions are independent if they are specified as parallel, i.e., they are parallel in $\mathbf{X}$. Actions are parallel if they are parallel in the implementation, i.e., in $\mathbf{Y}$.

The following example can be expressed by saying that independent actions in a specification may not be parallel in the implementation.

Given two automata $\mathbf{X}$, $\mathbf{Y}$, both with alphabet τ, suppose that there are refinements of $\mathbf{X}$ to $\mathbf{X}'$ and $\mathbf{Y}$ to $\mathbf{Y}'$, where $X' = X + U$ and $Y' = Y + U$, the meaning being that the set U is the state space of some temporarily used (and reset after use) resource like a scratch pad, or printer. Then the synchronous parallel product $\mathbf{X} \times \mathbf{Y}$ of $\mathbf{X}$ and $\mathbf{Y}$ may be refined to an automaton in which there is only one resource U whose use is scheduled between $\mathbf{X}$ and $\mathbf{Y}$. The state space would be $Z = XY + UY + XY + XU$, and the only letter acting on it would first apply $\langle \tau, \mathbf{1} \rangle$ until it lands in the third summand of the state space. Then, it would apply $\langle \mathbf{1}, \tau \rangle$ until it gets back to the first summand. The injection of XY as first summand of Z would then define a refinement, which would schedule the parallel action $\langle \tau, \tau \rangle$ to a sequence of actions of the form $\langle \tau, \mathbf{1} \rangle$ or $\langle \mathbf{1}, \tau \rangle$.

4.3. *Shutdown*

Consider a refined description of a system in which a new, destructive action can happen. This is a typical case of a sudden shutdown. We expect that the system can, at any time, be shut down, thus moving into a state which was impossible to reach before. In this case, the refinement space is formed by adding a single element, and a new letter to the alphabet; it sends to the new state any other state. The behaviour of the machine, if we ignore the shutdown state, is unmodified, which is exactly reflected in our definition of refinement.

4.4. *Choice*

Our refinement being a functor assigns to each action of the unrefined system a precise refinement. Hence it is not possible in our model to replace an action by two alternative actions even if two alternative actions may exist in the refined machine (such a thing would correspond to *two* different refinements). This accords with our view that machines, even asynchronous ones, are deterministic; the introduction of a choice in refinement is a non-determinism at the level of morphism. However, different choices can be identified by an abstraction morphism.

5. Comparisons

As we remarked in the introduction, our notion of refinement differs markedly from notions currently being considered in Petri nets and process algebra; rather, it is in the spirit of [CM88, AL87].

The definition which is conceptually closest to our approach is the broader definition of Petri net morphism given in [MM90], where a single Petri net transition can be mapped to an entire computation, possibly composed by many parallel steps. However, due to the freedom with respect to the monoidal product, the mapping is not dependent on the global state of the net.

In contrast to the situation in action refinement ([CvGG],[DGR]), in our model it is not at all necessary that a refinement of two parallel processors be parallel (§4.2) (and hence we can discuss scheduling of resources), or a refinement of conflicting processors be conflicting in all steps (§4.1) (and hence we can discuss refinements which limit non-parallelism to exactly those points where common resources are needed).

In contrast to Petri nets refinement ([BGV91]), we are unable to introduce a choice (§4.4) between actions to refine an action. This limitation simplifies considerably the theory but does not restrict its expressiveness.

References

[AL87] M. Abadi and L. Lamport. Composing specifications. In *Stepwise Refinement of Distributed Systems*, number 430 in LNCS, pages 1–41, 1987.

[BGV91] W. Brauer, R. Gold, and W. Vogler. A survey of behaviour and equivalence preserving refinement of Petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, number 483 in LNCS, 1991.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CvGG] I. Czaja, R. von Glabbeek, and U. Golz. Interleaving semantics and action refinement with atomic choice. Preprint.

[DGR] P. Degano, R. Gorrieri, and G. Rosolini. A categorical view of process refinement. In *Semantics: Foundations and Applications*, number 666 in LNCS.

[Elg75] C. Elgot. Monadic computation and iterative algebraic theories. *Studies in Logic and the Foundations of Mathematics*, 80:175–230, 1975.

[Hel90] A. Heller. An existence theorem for recursion categories. *Journal of Symbolic Logic*, 55(3):1252–1268, 1990.

[Knu73] D.E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[KWW] W. Khalil, E.G. Wagner, and R.F.C. Walters. Fixed-point semantics for programs in distributive categories. In preparation.

[Lyn87] N.A. Lynch. Multivalued possibility mappings. In *Stepwise Refinement of Distributed Systems*, number 430 in LNCS, pages 519–543, 1987.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. of the 2nd Joint Conference on Artificial Intelligence*, pages 481–489. BCS, 1971.

[MM90] J. Meseguer and U. Montanari. Petri nets are monoids. *Info. and Co.*, 88:105–155, 1990.

[SW93] N. Sabadini and R.F.C. Walters. On functions and processors: an automata theoretic approach to concurrency through distributive categories. Mathematics Report 93-7, Sydney University, 1993. Available by anonymous `ftp` at `ghost.dsi.unimi.it` in the directory `pub2/papers/sabadini`.

[Wal] R.F.C. Walters. *Categories and Computer Science*. Carslaw Publications (1991), Cambridge University Press (1992).

The role of Memory in Object-Based and Object-Oriented Languages

Eric G. Wagner
Wagner Mathematics
Old Albany Post Road
R 1 Box 445
Garrison, NY 10524 / USA
CSNET: Wagner@watson.ibm.com

Abstract

This paper introduces a algebraic memory model appropriate for programming languages with both ground types and objects, and related to an elementary inheritance, overloading, and class specification.

This paper reports on some the recent theoretical and practical results on programming constructs that came about as part of the continuing project to design, implement, and extend the programming language LD^3 (=Language for Data Directed Design) that I introduced at the first AMAST conference [4].

The main idea that I want to promote in this paper is that the proper context for talking about object-oriented and object-based programming is imperative rather than functional. That is, I will show why it is advantageous to view objects as parts of a state rather than as things-in-themselves. In particular I will show how this approach makes for rich variety of objects (or classes) and a simple approaches to inheritance and overloading.

Much of the theoretical work currently being done on OOL and OBL, e.g. [3, 1, 2], is done in a functional context wherein a method is viewed as a function on objects. In this paper we take we take a different approach based on a "memory" model, wherein the execution of a method both changes the state and returns a value. I first enunciated this approach in [5] where it was applied to produce a semantics for LD^3. The paper generalizes the treatment given there and examines some of its ramifications. This abstract gives only the first part of the story, a precise description of a particular form of inheritance – the full paper will also explore the role of overloading.

For any set K let $\mathbf{SStr}_K$ denote the free distributive category generated by K (the notation comes from the fact that this is also the category of strings-of-strings over K – see [4].

Definition 0.1 **Let K be a finite set, then K-*state* μ is given by the following data**

$I_\mu = \langle I_\mu(k) \mid k \in K \rangle$, **a K-indexed family of sets.**

$V_\mu = \langle V_\mu(k) \mid k \in K \rangle$, **a K-indexed family of sets.**

$\mu = \langle \mu(k) : I_\mu(k) \rightarrow V_\mu(k) \mid k \in K \rangle$, **a K indexed family of mappings.**

If we view I_μ and V_μ as functors from the discrete category K to the category **Set** of sets and functions, then is μ is a natural transformation.

Given K-states μ and μ' we define a morphism $\eta : \mu \to \mu'$ to be a pair of injective natural transformations, $\langle \alpha : I_\mu \to I_{\mu'},\ \beta : V_\mu \to V_{\mu'} \rangle$ such that $\mu' \bullet \alpha = \beta \bullet \mu$. Here α injective means α_k is injective for every $k \in K$. Let $\mathbf{ST}_K$ denote a category of K-states. □

The rough intuition is that in a K-state $\mu = \langle I_\mu, V_\mu, \mu \rangle$ what is specified is a K-indexed set I_μ of entities (locations, objects) each entity having a value in the K-indexed set V_μ as specified by the K-indexed family of functions μ. More specifically, the entities belonging to class $k \in K$ are the elements of the set $I_\mu(k)$, their possible values are the elements of the set $V_\mu(k)$ and there specific values in the state are given by the mapping $\mu_k : I_\mu(k) \to V_\mu(k)$. The morphisms in the category $\mathbf{ST}_K$ capture the notion of "substate".

The next definition extends the idea of "entities" with values by extending each set $I_\mu(k)$ to include an additional entity without a value. We shall call the additional entity of class k the *null-entity* of class k. Such null-entities provide a means for dealing with constructs as such as null-pointers.

Definition 0.2 From I_μ we define a functor $O_\mu : K \to \mathbf{Set}$, $k \mapsto I_\mu(k) + 1$. Given a natural transformation $\alpha : I_\mu \to I_{\mu'}$ we extend it to a natural transformation from $\alpha^{!} : O_\mu \to O_\mu$ by taking $\alpha^{!}_k = \alpha_k + 1$ for all $k \in K$. It what follows we shall generally omit the " $!$ " and use the same notation for both natural transformations. □

Since $\mathbf{SStr}_K$ is the free distributive category generated by K it follows that for any state μ, the functors I_μ, V_μ and O_μ extend, canonically, to (respective functors $\widehat{I}_\mu$, $\widehat{V}_\mu$ and $\widehat{O}_\mu$) from $\mathbf{SStr}_K$ into **Set**. We will generally omit the "$\widehat{\ }$" in future uses of these functors.

Definition 0.3 For each $v \in (K^*)^*$ define $U^v : \mathbf{ST}_K \to \mathbf{Set}$, $\mu \mapsto O_\mu(v)$ and $\langle \alpha, \beta \rangle \mapsto \alpha_v$. to be the functor with the above indicated object- and morphism-parts. □

Definition 0.4 For each $v \in K^*$ we define a category $\mathbf{O}^v_K$ with, as objects, all pairs $\langle \mu, e \rangle$ where μ is a state and $e \in U^v(\mu) = O_\mu(v)$, and, as morphisms from $\langle \mu, e \rangle$ to $\langle \mu', e' \rangle$, those morphisms $\eta : \mu \to \mu'$ such that $U^v(\eta)(e) = e'$ (so, if $\eta = \langle \alpha, \beta \rangle$ then $\alpha_v(e) = e'$).

For $v \in K^*$, let $\Pi = \Pi^v : \mathbf{O}^v_K \to \mathbf{ST}_K$, $\langle \mu, e \rangle \mapsto \mu$ and $\eta \mapsto \eta$. □

Definition 0.5 Let $F : \mathbf{O}^u \to \mathbf{O}^v$ be a partial functor, then for each $k \in K$ define $\overline{F}_k : \mathbf{O}^v_K \to \mathbf{Set}$, $\langle \mu, e \rangle \mapsto O_\mu(k)$, and $\langle \alpha, \beta \rangle \mapsto \alpha_k$. Observe that, $\overline{F}_k = U^{(k)} \bullet \Pi^v \bullet F$. □

Definition 0.6 By an $\mathbf{ST}_K$*-operation* of arity $\langle v, w \rangle \in (K^*)^* \times (K^*)^*$ we mean a partial functor $F : \mathbf{O}^v_K \to \mathbf{O}^w_K$ equipped with an injective natural transformation $\iota^{F,k} : \overline{\mathbf{1}}_k \to \overline{F}_k$ for each $k \in K$, where $\mathbf{1}_k$ denotes the identity functor on $\mathbf{O}^{(k)}_K$. □

The idea here is that an $\mathbf{ST}_K$-operation, F, of arity $\langle u, v \rangle$ is a possible semantics for "functions" with formal parameters specified by the string u and returning results specified by the string v where "side-effects" are allowed, i.e., execution of a "function" can result in a change of state as well as the return of a value.

The functorality of F captures a somewhat more subtle point, mainly the intuitive idea that if a "function" is defined for a given state μ and input e and μ' is an "extension" of μ then the function is also defined for μ' and e and, indeed, does essentially the same thing then as it did before. The mathematics makes a slightly weaker, but more precise statement.

The requirement that we have an injective natural transformation $\iota^{F,k} : \overline{\mathrm{I}}_k \rightarrow \overline{F}_k$ for each $k \in K$ can be interpreted as saying that the execution of F "preserves entities", i.e., that if $F(\langle \mu, e \rangle) = \langle \mu', e' \rangle$ then, roughly speaking, $I_\mu(k) \subseteq I_{\mu'}(k)$ for every $k \in K$.

Definition 0.7 Given $\mathbf{ST}_K$-operations $\langle F, \iota^F \rangle : \mathbf{O}_K^u \rightarrow \mathbf{O}_K^v$ and $\langle G, \iota^G \rangle : \mathbf{O}_K^v \rightarrow \mathbf{O}_K^w$, we define their composite, $\langle G, \iota^G \rangle \bullet \langle F, \iota^F \rangle : \mathbf{O}_K^u \rightarrow \mathbf{O}^w$ to be $\langle G \bullet F, \iota^{GF} \rangle$ where, for each $k \in K$, and $\langle \mu, e \rangle \in Obj(\mathbf{O}_K^u)$, $\iota^{GF,k} = \iota^{G,k}_{F(\langle \mu, e \rangle)} \bullet \iota^{F,k}_{\langle \mu, e \rangle}$. □

Proposition 0.8 *The* $\mathbf{ST}_K$*-operations form a category,* $\mathbf{Op}_K$*, with the above defined composition and with the identity morphism,* $id_v : \mathbf{O}_K^u \rightarrow \mathbf{O}_K^u$ *being* $\langle 1_{\mathbf{O}_K^u}, \langle 1_{k,\mu} : O_\mu(k) \rightarrow O_\mu(k) \mid k \in K, \mu \in \mathbf{ST}_K \rangle \rangle$.

Definition 0.9 Let $k, k' \in K$, then by the *replacement function* for k and k' we mean the function $r_{k,k'} : K \rightarrow K$ such that $r_{k,k'}(j) = k$ if $j = k'$ and $r_{k,k'}(j) = j$ otherwise.

□

Definition 0.10 A *smooth coercion* from k' to k is a natural transformation $c : U^{(k')} \rightarrow U^{(k)}$. □

Proposition 0.11 *If* $k, k' \in K$, $u, v \in K^*$ *such that* $r_{k',k} \bullet u = v$ *and* c *is a smooth coercion from* $U^{(k')}$ *to* $U^{(k)}$ *then there is a functor* $\vec{c} : \mathbf{O}_K^u \rightarrow \mathbf{O}_K^v$ *such that for all* $\langle \mu, e \rangle \in Obj(\mathbf{O}_K^u)$, $\vec{c}(\langle \mu, e \rangle) = \langle \mu, e' \rangle$ *(no change in the state!) where* $e'_i = c(e_i)$*if* $u_i = k'$*, and* $e'_i) = e_i$ *if* $u_i \neq k'$*. Furthermore, this functor* $\vec{c}$ *is an* $\mathbf{ST}_K$*-operation when equipped with the identity natural transformation* $\overline{\mathrm{I}}_k \rightarrow \overline{\vec{c}}_k$ *for each* $k \in K$.

Proposition 0.12 *If* $k, k' \in K$, $u, v \in K^*$ *such that* $r_{k',k} \bullet u = v$ *and* c *is a smooth coercion from* $U^{(k')}$ *to* $U^{(k)}$ *then there is an induced mapping*

$$\begin{array}{c} \mathbf{Op}_K(v, w) \rightarrow \mathbf{Op}_K(u, w) \\ f \mapsto f \bullet \vec{c}. \end{array}$$

We now apply these ideas in a more concrete setting.

Definition 0.13 A *class-system* is specified by the following data:

G, called the set of names for *ground classes.*

D, called the set of names for *defined classes.* Let $K =_{def} G + D$.

$\iota : D \rightarrow (K^*)^*$, called the *form function.*

$\mathcal{G}$, a G-sorted algebra, called the *algebra of ground operations* with some signature Γ.

Given a class-system $\mathcal{K} = \langle G, D, \iota, \mathcal{G} \rangle$ a *basic state*, μ, for $\mathcal{K}$ consists of

$I_\mu : K \rightarrow \mathbf{Set}$. For our current purposes let us assume that for each $k \in K$ $I_\mu(k) = \{\langle j, k \rangle \mid j = 1, \ldots, n_k\}$for some $n_k \geq 0$.

Let $V_\mu : K \rightarrow \mathbf{Set}$ such that $g \mapsto \mathcal{G}_g$ and $d \mapsto O_\mu(\iota(d))$.

$\mu : I_\mu \rightarrow V_\mu$ a natural transformation.

Finally, we restrict ourselves to morphisms $\langle \alpha, \beta \rangle : \mu \rightarrow \mu'$ between states in which, for each $d \in D$, $\beta_d = \alpha_{\iota(d)}$, and, for each $g \in G$, $\beta_g = 1_{\mathcal{G}_g}$. □

Now let us restrict our attention to $\mathbf{ST}_K$-operations $\langle F, \iota^F \rangle$ in which the $\iota^{F,k}_{(\mu,\epsilon)}$ are inclusion mappings. We can show that there are more than encugh such operations to form a programming language (see the $\mathbf{LD}^3$ papers).

Proposition 0.14 *Let $k, k' \in K$, then if there exist $u, v \in (K^*)^*$ and an isomorphism $\rho : \iota(k') \cong ((k) \times u) + v$ then there exists a corresponding smooth coercion $c_\rho : U^{(k')} \to U^{(k)}$.*

1. There is always a trivial example: Take $u = \mathbf{0}$, the empty string-of-strings, and take $v = \iota(k')$, then (because, for any u and v, $u \times \mathbf{0} = \mathbf{0}$, and $\mathbf{0} + v = v$) we trivially have $iota(k') \cong ((k) \times \mathbf{0}) + \iota(k')$. I claim that the corresponding smooth coercion is the one which, for each state μ is given by the mapping

$$\begin{gathered} O_\mu(k') \to O_\mu(k) \\ \langle j, k' \rangle \mapsto \langle 0, k \rangle. \end{gathered}$$

2. Assume $\iota : D \to (K^*)^*$ is such that, for every $d \in D$, $\iota(d) = (k_{k,1} \cdots k_{k.n_k})$, i.e. it consists of a single string of length n_k. This is the case in object oriented languages where the state is given by the values of a set of instance variables. When this is the case, it is easy to see that, if $d \in D$ and $k \in K$ such that $\iota(k') = (k_{k',1} \cdots k_{k',n_{k'}})$and $k = k_{k',i}$ for some i, then we have an isomorphism

$$\mathbf{i}_\mu : \iota(k') \cong (k) \times (k_{k',1} \cdots k_{k',i-1} \cdot k_{k',i+1} \cdots k_{k',n_{k'}})$$

   The desired smooth coercion here the one which for each state μ is given by the mapping $\mathbf{O}_\mu(k') \to \mathbf{O}_\mu(k)$ taking $x \in I_\mu(k')$ to $\mu(x)_i$.

3. We claim that the archtypical example of inheritance – the inheritance of the **move** operation on the class **dots** by the "subclass" **colored_dots** – is an example of just such a smooth coercion. We will give a fuller treatment this, and other examples, in the full paper.

References

[1] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 4:471–522, 1985.

[2] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. Technical Report LIENS - 92 -4, LIENS, Laboratoire d'Informatique de l'Ecole Normale Superiure, February 1992.

[3] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th POPL*, pages 109–124. ACM, 1990.

[4] Eric G Wagner. An algebraically specified language for data directed design. *Theoretical Computer Science*, 77:195–219, 1990.

[5] Eric G. Wagner. Generic classes in an object-based language. In *Recent Trends in Data Type Specification*, pages 330 – 344. LNCS 655, Springer Verlag, 1992.

Abstract and Concrete Objects – An Algebraic Design Method for Object-Based Systems

Ruth Breu [1], Michael Breu [2]

This paper demonstrates the design of an object-based system using algebraic specification techniques. The flexibility of algebraic specifications allows the system to be described at any stage of the design – starting from a descriptive specification and ending at a constructive specification. The latter one is a specification at the level of a program, comprising concrete data representations and machine-executable algorithms.

Since we are developing an object-based system, the algebraic target specification in particular is a specification of *objects*. In our framework objects are entities with a unique identity and an evolving internal state which can be manipulated by the outside through a set of operations (commonly called *methods*). In this paper we are restricting ourselves to environme exactly one active object at a time. Hence, the resulting specification can be easily transl a typed sequential object-oriented program such as an Eiffel or C++ program.

In our opinion, the notion of objects is too concrete to be the basis for the whole design. In particular, object states, object sharing and side effects of methods are facilities which are tightly connected with the notion of objects but encounter aspects of abstractness and implementation independence.

Therefore, we suggest a design method which is based on a two-tiered paradigm of object specification. The early stages of the design rely on a notion of *abstract objects*. Abstract objects are stateless values on which a set of functions can be applied yielding other abstract objects. The specification of abstract objects is based on an external view, stating the behaviour of the functions. In particular, abstract objects are independent of data representations and do not have states.

In later stages of the design, abstract objects are implemented by a state based object description. These state dependent objects are called *concrete objects*. Concrete objects exist in *object environments* in which one object may refer to other concrete objects.

During the transition from abstract to concrete objects, a formal notion of implementation has to ensure that the correctness of the system description is preserved. We take the approach of [Breu R 91] and relate abstract and concrete objects by abstraction functions mapping each state of a concrete object to an abstract object. While in [Breu R 91] abstraction functions connect algebraic specifications with object-oriented programs in a model based theory, in this paper the axiomatic framework is not left. Following the idea of [Breu M 90], the abstraction functions

[1] Technische Universität München, Institut für Informatik, Postfach 20 24 20, D - 8000 München 2

[2] Siemens Nixdorf Informationssysteme AG, European Methodology and System Center,
Otto-Hahn-Ring 6, D - 8000 München 83

are part of the algebraic specification and hence enable reasoning at the level of a formal calculus.

Our approach goes beyond related approaches since it supports the specification of both abstract and concrete objects. In this respect, our framework can be considered as an extension of approaches which pursue the specification of concrete objects ([Goguen, Meseguer 87], [America, de Boer 90]). A similar separation into state dependent and state independent objects together with abstraction functions can be found in [Wing 87] and [America 90]. Unlike these approaches, our framework is based on a uniform logic environment in which both abstract and concrete objects are specified and proofs are performed.

As syntactic and semantic framework, we rely on the algebraic specification language SPECTRUM ([Broy et al. 91]). This specification language provides facilities like the specification of partial functions and higher-order functions, admitting formulas of a general predicate logic.

We will illustrate our ideas by the common example of binary trees. The full version of this paper will contain a larger case study. This case study deals with the implementation of the most general unifier of terms based on an object structure which relies on a shared representation of terms, i.e. a representation by dags.

The Specification of Abstract Objects

A primary goal in the first stage of our design method is the identification of abstract objects together with the abstract specification of their behaviour. We model abstract objects by values of some sort (called the *object sort*) in an algebraic specification.

Abstract objects in our example are binary trees (of object sort *Tree*). Binary trees are as usually attached with two constructors ε: *Tree* and *node: Tree* × *Nat* × *Tree* → *Tree*. Moreover, *left, right: Tree* → *Tree* and *label: Tree* → *Nat* denote the projections to the first and second subtree and to the label of the root, respectively. The related specification is straightforward. It can be found for instance in [Wirsing 90].

The Specification of Concrete Objects

Each concrete object consists of

- a unique identity
- an evolving state which may refer to other concrete objects.

Concrete objects thus do not exist in an isolated setting, but in an object *environment*. Object environments are collections of concrete objects which are connected by a network of references. This includes the facility of references to common subobjects (*object sharing*).

In our example, we implement the abstract tree objects by concrete objects which form a dag structure. Figure 1 depicts an environment of two objects representing the abstract tree *node(node(ε, 2, ε), 1, node(ε, 2, ε))*.

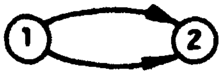


Figure 1

In object-oriented languages object environments and object identities are implicitly given. In a framework in which properties are proved formally, an explicit modelling is advantageous in order to keep the logic simple.

We model concrete objects of object sort *s* by an algebraic specification containing the following features.

- A sort Id_s describes the set of object identities.
- A sort $State_s$ describes the set of object states.
- A sort *Env* describes the set of object environments. This set is characterised by associations of object identities with object states.
- Methods are modelled by functions f: $Env \rightarrow Env$ on object environments. Additional parameters may refer to concrete objects in the environment or to basic values.

It has to be noted that the specification of object identifiers and environments does not necessarily be a specification of a low-level pointer structure. More abstractly, object identifiers can be conceived as identifying keys and object environments as databases relating keys with object states.

The Transition from Abstract to Concrete Objects

We relate abstract and concrete objects by abstraction functions. Each abstraction represents a particular state in the lifetime of a concrete object by a stateless value. Formally, the abstraction is a function *abstr* mapping environments and object identities to values of the abstract object sort. An application of the abstraction function *abstr* in our example of binary trees is sketched in figure 2. Object identities (of sort Id_{Tree}) are indicated by an arrow in the given environment.

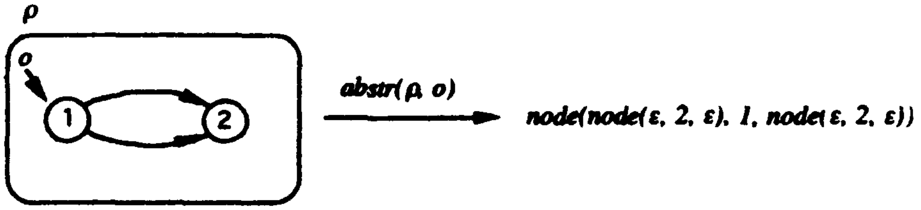
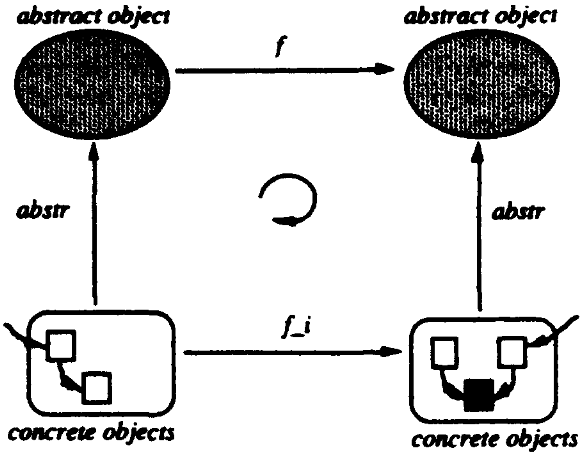


Figure 2

An important property which we require of an abstraction function *abstr* is its compatibility with the functional behaviour of objects. This *homomorphism property* is characterised in the following diagram.



The above diagram commutes for any operation f on abstract objects corresponding to a method f_i on concrete objects. Thus, abstraction functions between concrete and abstract objects are homomorphisms augmented by a notion of states. These extended homomorphisms have been called *state based homomorphisms* in [Breu R 91].

More precisely, the implementation of abstract objects of object sort s consists of the following four steps. We assume that a specification of object environments (of sort Env), object identities (of sort Id_s) and object states (of sort $State_s$) is already given.

I1. Implementation of the functions associated with abstract objects

Each function f in the abstract specification is implemented by a method f_i working on environments. Each occurrence of sort s in the arity of f corresponds to the sort Id_s in the arity of f_i. In this way, we obtain for instance the arities $node\_i\colon Env \times Id_{Tree} \times Nat \times Id_{Tree} \rightarrow Env \times Id_{Tree}$ and $left\_i\colon Env \times Id_{Tree} \rightarrow Env \times Id_{Tree}$ in our example of binary trees.

Related axioms describe the behaviour of these functions. In our example, ε_i and $node\_i$ are methods which create new objects, $left\_i$, $right\_i$ and $label\_i$ do not change the given environment, i.e. are constant on the first argument.

I2. Abstract specification of the abstraction function

In this step we introduce the abstraction function

$$abstr\colon Env \times Id_s \rightarrow s$$

together with axioms specifying the homomorphism properties. Since, in general, these properties are too strong to be valid in the set of all object environments, we introduce a constraint on environments

$$I\colon Env \rightarrow Bool.$$

For each function $f\colon s \rightarrow s$ on abstract objects we introduce the axiom

$$\text{-- } ABSTR\_AX \text{ --} \quad \forall \rho\colon Env; x\colon Id_s \ in \quad I(\rho) \rightarrow abstr(f\_i(\rho, x)) = f(abstr(\rho, x)).$$

Axioms related with functions with general arity $f\colon s_1 \times \ldots \times s_n \rightarrow s_0$ are obtained in an analogous way.

Additional axioms may describe abstractly the side effect of the functions *f_i* on the argument objects based on the abstraction function. Note that at this stage the boolean function *l* does not have related axioms, i.e. it is totally loose.

I3. Constructive specification of the abstraction function

The axioms in step I2 describe the function *abstr* in a non-constructive way. In step I3, axioms have to be introduced which define this function explicitly based on the structure of object environments and object states. Moreover, the loose specification of the constraint function *l* on environments has to be concreted. The specification of this function is deferred to this stage since it is tightly connected with the idea of the implementation of the abstraction function.

In our example, the abstract tree object related with a concrete tree object is obtained by collecting the node information along the trace of references in the environment. The constraint $l(\rho)$ holds if the environment ρ forms a dag structure, i.e. does not contain cyclic networks of objects.

I4. Proofs of correctness

In the last step, the soundness of step I3 with respect to the abstract axioms of step I2 has to be proved. This means that the homomorphism axioms *ABSTR_AX* have to be converted into theorems in the theory of the specification of step I3.

After the elimination of the non-constructive axioms of step 2, the developed target specification should contain axioms which describe algorithms related with

- the functions *f_i* on concrete objects and
- the abstraction function *abstr*.

At this stage, the development has reached a level at which the transition to a machine-executable program does not change the level of abstraction.

Conclusion

A main advantage of our design method is the gain of abstractness compared to approaches which are based on the specification of state dependent objects. In particular, our approach supports the separated development of algorithms and data representations.

A second main advantage of our approach is the uniform logic framework of the design. Through the explicit specification of object identifiers and object environments, the simple logic calculus of the functional framework can be applied. Nevertheless, it has to be stressed that the explicit modelling of these state based features does neither have effects on the style nor on the expressiveness of object specifications.

References

[America 90] P. America: ***Designing an object-oriented programming language with behavioural subtyping***. In: J.W. de Bakker et al. (eds.): Foundations of Object-Oriented Languages, Proc. REX School/Workshop, The Netherlands, May/June 1990. Lecture Notes in Computer Science **489**, Springer, 1991, 60-90

[America, de Boer 90] P. America, F. de Boer: *A Proof System for Process Creation*. In: Proc. IFIP TC 2 Working Conference on Programming Concepts and Methods, April 1990

[Breu M 90] M. Breu: ***Development of Implementations***. In: PROgram Development by SPECification and TRAnsformation, Volume I. Esprit Project 390 PROSPECTRA, Report M2.2.S4 - R - 11.0, 1990 (to appear in the series of Springer Lecture Notes in Computer Science)

[Breu R 91] R. Breu: ***Algebraic Specification Techniques in Object Oriented Programming Environments***. Lecture Notes in Computer Science **562**, Springer, 1991

[Broy et al. 91] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, K. Stølen: ***The Requirement and Design Specification Language SPECTRUM – An Informal Introduction, Version 0.3***. Report TUM-I9140, Technische Universität München, 1991

[Goguen, Meseguer 87] J.A. Goguen, J. Meseguer: ***Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics***. In: B. Shriver, P. Wegner (eds.): Research Directions in Object-Oriented Programming. MIT Press, 1987, 417-477

[Wing 87] J. M. Wing: ***Writing Larch Interface Language Specifications***. ACM Transactions on Programming Languages and Systems **9**:1, 1-24 (1987)

[Wirsing 90] M. Wirsing: ***Algebraic Specification***. In: J. van Leeuwen (ed.): Handbook of Theoretical Computer Science. Elsevier Science Publishers, 1990

Towards an Algebraic Theory of Inheritance in Object Oriented Programming

Xue-Miao Lu and Tharam S. Dillon
Department of Computer Science and Computer Engineering,
La Trobe University, Bundoora, Victoria, AUSTRALIA 3083
Email: {lu, tharam}@latcs1.lat.oz.au

Abstract

In this paper an approach is proposed to the algebraic specification of classes and inheritance in object oriented programming, using the notion of algebraic implementation of abstract data types.

1 Introduction

Application of the algebraic theories to computer science and software technology has been widely studied. As the object oriented paradigm[3,5,9,13,15] has become increasingly important as a new software engineering methodology, attempts have been made to give a rigorous mathematical foundation for object oriented systems using the algebraic theories(e.g.,[1,4,6,7,10,11]). In particular, algebraic models of inheritance have been proposed, e.g.,[4]. However, the existing models use conventional notions such as *signature morphism* and thus are not wide enough to provide the representation of incremental inheritance.

In object oriented programming, the central concept is *object*. An object has an identifier, attributes and methods. An important feature in object oriented programming is *classification*, incorporating the notion of *encapsulation*. Classification organizes objects into *classes*. Attributes and methods of an object are defined in the class. Inheritance is an important feature of object oriented systems, providing a mechanism for defining attributes and methods for a new (sub)class from definitions of in (super)class. In a class specification, there is an interface part which, by providing the attributes and methods, designates how to build and manipulate objects of the class[9,13,15]. This interface is an (abstract) *object type* specification[12]. For some classes, this part characterizes all the features of a class. For others, however, it does not fully characterize a class and another part is used that concerns implementation of the current type[9,13,15]. We call this the *implementation part*. These two parts form an *implementation specification* in terms of algebraic data types.

Because of lack of space, we only discuss some basic ideas in this paper and refer to [12] for further details. Attributes and methods (functions) in a class are grouped into kinds: *meta-methods, instance attributes, instance methods, class attributes, class methods*, and *shared instance attributes*[5,9]. Here we only consider the first three features of them. Higher types are used for specifications of object types. Inheritance is considered on two levels: on types and on classes. Generally, inheritance supports *incremental modification, renaming, overriding*, and *specialization*; and inheritance can be *single* or *multiple*. In this paper, we only discuss single inheritance supporting incremental modification, assuming that an inherited component has the same name as in the source. The loose approach has been shown in [12] to be especially appropriate for the semantics of inheritance, including the significant point that an object of an inheriting type is also an object of its super type.

2 Object Types and Inheritance

We assume familiarity with the basic notions from the equational algebraic specification. An *object type* $\sigma = O(k_1 : s_1, \ldots, k_n : s_n)$ consists of pairs $k_i : s_i$, which we will call *components* of the object type. Denote $K^\sigma = \{k_1 : s_1, \ldots, k_n : s_n\}$. K^σ is divided into two disjoint subsets, K^σ_{mt} and K^σ_{in}, with $K^\sigma_{in} = \{k_m : s_m, \ldots, k_n : s_n\}$, $1 < m < n$, such that (1) for each method in K^σ_{mt}, its coarity is exactly σ. In particular, there can be a *construction method* $\mathrm{tup}^{w \to t} \in K^\sigma_{mt}$ with $w = s_m \times \ldots \times s_n$

and a constant operation symbol create$^{\perp \to \sigma}$; and (2) for each method in K^{σ}_{in}, at least one of its argument types, or the coarity, is an attribute type in K^{σ}_{in}. Components in K^{σ}_{mt} are *meta-methods*, and those in K^{σ}_{in} are *instance attributes* and *instance methods*. A component $k_i : s_i$ has a name k_i and a type s_i. When an object type τ inherits σ, we write $\sigma \rightsquigarrow \tau$.

A *typed signature* Σ is an S-sorted signature $\Sigma = (S, \Sigma)$ such that for each object type σ, every method name k_i in σ is an operation symbol, and for each $k_i \in K^{\sigma}_{in}$, there is a unary *named projection* operation symbol $k_i \in \Sigma_{\sigma, s_i}$; and if an inheritance relation $\sigma \rightsquigarrow \tau$ exists, then for each meta-method $\alpha : s_1 \times \ldots \times s_q \to \sigma$ for σ, there is a meta-method $\beta : s_1 \times \ldots \times s_q \times \ldots \times s_p \to \tau$ for τ, $p \geq q$, and β is said to be *compatible* with α. A *typed specification* HTSP=(Σ, E) consists of a typed signature Σ and a set E of axioms. A typed specification HTSP is also written HTSP=(S, Σ, E).

An *algebra* A for a typed specification HTSP=(S, Σ, E) is a specification algebra, with a *carrier* s^A for each sort $s \in S$, such that for each method name k_i of type t, there is a uniquely assigned method in t^A, denoted by k_i^A, or simply k_i; and such that for every inheritance relation $\sigma \rightsquigarrow \tau$, each component $k : s$ in K^{σ}_{in}, $k^A : \tau^A \to s^A$ is also a projection operation. In either of the cases, we say that k is *inherited*, and σ is said to be *(weakly) inherited by* τ.

Let $\sigma \rightsquigarrow \tau$ in HTSP. For an HTSP-algebra A and meta-method β compatible with α as mentioned above. α is said to be *inherited to* β in A if for $a_i \in s_i^A$, $1 \leq i \leq p$, $k_i(\beta(a_1, \ldots, a_q, \ldots, a_p)) = k_i(\alpha(a_1, \ldots, a_q))$ for each instance component name k_i. σ is said to be *strongly inherited* by τ in A if each meta-method k_i in σ is inherited in A. In this case, α can be *compatibly adapted* to τ in A by defining $\alpha(a_1, \ldots, a_q) = \beta(a_1, \ldots, a_q, c_{q+1}, \ldots, c_p)$ for fixed elements $c_j \in s_j^A$, $q < j \leq p$.

Proposition 2.1 *Let $\sigma \rightsquigarrow \tau$ in the typed specification HTSP. Assume that σ is not an attribute type for any other object type in HTSP. For any HTSP-algebra A, let B be obtained from A by replacing σ^A by $\sigma^A \cup \tau^A$, then B is an HTSP-algebra.* ▮

Proposition 2.2 *With the conditions above, for any HTSP-algebra A in which σ is strongly inherited by τ, let B be obtained from A by replacing σ^A by τ^A, then B is an HTSP-algebra after compatibly adapting α to τ in A.* ▮

In both propositions, B being an HTSP-algebra means that objects of type τ are objects of σ. This is an important feature in the object oriented framework.

3 Implementation of Object Types

In this section we briefly discuss the notion of implementation. We utilize existing notions of implementation in the literature[2,8,14] and integrate the distinctness of object orientation.

Let HTSP and SPEC=(S, Σ, E) be the typed specifications of σ and τ, respectively. The basic idea of implementing σ by τ (or HTSP by SPEC) is to use the features in SPEC to describe those in HTSP. In case several sorts, $s_1, \ldots, s_p$, in SPEC are used to describe one sort t in HTSP, we denote the sequence of sorts by $< s_1, \ldots, s_p >$ and call it a *joint sort*, and it is essentially a product type. Each s_i in a joint sort is associated with a fixed attribute of type s_i. Two or more attributes, $k_1, \ldots, k_p$ in SPEC can be used to describe one attribute k in HTSP, and we denote this sequence by $< k_1, \ldots, k_p >$ and call it a *joint attribute*. Moreover, methods are needed for manipulating these joint attributes. We will call these methods *compound methods*. A compound method involves one or more existing methods in SPEC and consists of terms $(t_1, t_2, \ldots, t_u)$ of appropriate types from $T_{\Sigma}(X)$. This is an ordered sequence and the effect is equivalent to the sequential actions of these component terms as operations. With the introduction of a joint sort $s =< s_1, \ldots, s_p >$, a meta-method whose argument types include all the types in s can be rewritten by substituting s for the occurrences of $s_1, \ldots, s_p$.

A joint sort or a joint attribute exists only functionally, that is, it is not a component in a specification. Instead, a joint sort means that several existing sorts will be involved for a single action by a (compound) method. Similarly, a joint attribute of a joint type means that several attributes will be involved in an action by a (compound) method. In contrast, a compound method is a component of the object type. A compound method can be defined on a joint attribute and thereby it may access the attributes or change the values of the attributes involved in the joint attribute when the compound method is invoked.

To implement HTSP by SPEC, we follow the three stages, *synthesis*, *restriction*, and *identification* in [8]. The first stage is to enrich SPEC to EnSP. An *enrichment* of SPEC is a typed specification EnSP obtained from SPEC by adding attributes of the existing types (rather than σ) in SPEC, defining a set of joint sorts and a set of joint attributes to SPEC, adding a number of compound methods, and accordingly the operations for the new components. We do not use the ***sort implementing operations***, rather we use the implementation morphism, in a similar manner to [14]. In the second stage, EnSP is restricted to EnSP_{impl} by deleting the methods which are not used directly in simulation. Finally representatives are selected from a given EnSP_{impl}-algebra A using a congruence, to simulate an HTSP-algebra. In what follows, we always assume that HTSP, HTSP1, SPEC and SPEC1 are typed specifications of σ, σ_1, τ and τ_1, respectively, and $\text{HTSP}=(S,\Sigma,E)$ and $\text{HTSP1}=(S1,\Sigma1,E1)$.

An ***implementation morphism*** from HTSP to HTSP1 is an injective mapping h from Σ to $\Sigma1$ such that $h(\sigma)=\sigma_1$; and for any operation symbol $\alpha : s_1 \times \ldots \times s_q \to s$, if α is a meta-method for σ, then $h(\alpha)$ is of the form $h(s_1) \times \ldots \times (s_q) \times t_1 \ldots \times t_p \to \sigma_1$, otherwise, $h(\alpha)$ is of type $h(s_1) \times \ldots \times (s_q) \to h(s)$. If $h(s)=s$ for each $s \neq \sigma$ that is not a meta-method sort in σ, we call h an ***inheritance morphism***.

An implementation morphism is a generalization of a signature morphism in the conventional sense since it does not necessarily preserve the meta-methods. The implementation morphism *impl* maps each sort s of HTSP to a sort or to a joint sort in EnSP, an attribute to an attribute or a joint attribute, and a method to a method or a compound method.

An ***implementation*** of $\text{HTSP}=(\Sigma,E)$ of σ ***on*** SPEC of τ is given by an enriched specification EnSP of SPEC and an implementation morphism *impl* from HTSP to EnSP, and denoted by (HTSP, *impl*, EnSP, SPEC). A ***model*** M of the implementation (HTSP,*impl*,EnSP,SPEC) is quadruple $(A, impl^M, B, \equiv_{impl})$ consisting of an HTSP-algebra A, an EnSP_{impl}-algebra B and an injective homomorphism $impl^M$ from A to a congruence $B/\equiv_{impl}$ of B, where EnSP_{impl} is the typed specification obtained from EnSP by deleting the methods in EnSP which are not within the image of *impl*.

The definition of implementation is a partial one in the sense that we do not require that every HTSP-algebra can be represented by an EnSP_{impl}-algebra. If EnSP=SPEC in (HTSP, *impl*, EnSP, SPEC), we say that HTSP is implemented ***by*** SPEC, and denote it by (HTSP, *impl*, SPEC). There can be multiple ways for implementing an object type on (by) another in that a compound method may be composed using a different set of terms. In addition, an implementation can have many models. It is easy to see that if SPEC1 is an enrichment of SPEC and SPEC2 an enrichment of SPEC1, then SPEC2 is an enrichment of SPEC; and if *impl* is an implementation morphism from HTSP to SPEC1 and $impl_1$ an implementation morphism from SPEC1 to SPEC2, then $impl_1 \circ impl$ is an implementation morphism from HTSP to SPEC2.

Proposition 3.1 *(Composition of Implementations) If (HTSP,impl,EnHT1,HTSP1) and (HTSP1, $impl_1$,EnSP,SPEC) are implementations, then (HTSP,$impl'_1 \circ impl$,EnEnSP,SPEC) is an implementation and the diagram in Figure 1(a) commutes on $K_{in}^{\sigma_1}$, i.e., for $k \in K_{in}^{\sigma_1}$, $f_1 \circ impl_1(k) = impl'_1 \circ g(k)$, where EnEnSP is an enrichment of EnSP constructed in a natural way along the construction of EnHT1 from HTSP1, and $impl'_1$ is an extension of $impl_1$; and for $impl(t) =< t_1, \ldots, t_q >$ with $impl_1(t_i) =< t_{i,1}, \ldots, t_{i,p_i} >$, $impl'_1 \circ impl(t) =< t_{1,1}, \ldots, t_{1,p_1}, \ldots, t_{q,1}, \ldots, t_{1,p_q} >$.*

Moreover, (EnHT1_{impl},$impl'_2$,EnEnSP,SPEC) is an implementation of EnHT1_{impl} on SPEC, where $impl'_2$ is the restriction of $impl'_1$ on EnHT1_{impl}. And if $(A, impl, B, \equiv_{impl})$ is a model for (HTSP, impl, EnHT1, HTSP1) and $(B, impl'_2, C, \equiv_{impl'_2})$ is a model for (EnHT1_{impl}, $impl'_2$, EnEnSP, SPEC), then $(A, impl'_2 \circ impl, C, \equiv_{impl} \circ \equiv_{impl'_2})$ is a model for (HTSP, $impl'_2 \circ impl$, EnEnSP, SPEC), where $C/(\equiv_{impl} \circ \equiv_{impl'_2})=(C/\equiv_{impl'_2})/\equiv_{impl}$. ▌

(a)

$$\begin{tikzcd}
 & \text{HTSP1} \arrow[r, "g"] \arrow[d, "impl_1"] & \text{EnHT1} \arrow[d, "impl'_1"] & \text{HTSP} \arrow[l, "impl"'] \\
\text{SPEC} \arrow[r, "f"] & \text{EnSP} \arrow[r, "f_1"] & \text{EnEnSP} &
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
 & \text{HTSP} \arrow[r, "h"] \arrow[d, "impl"] & \text{HTSP1} \arrow[d, "impl_1"] \\
\text{SPEC} \arrow[r, "f"] & \text{EnSP} \arrow[r, "f_1"] & \text{EnSP1}
\end{tikzcd}$$

Figure 1.

Proposition 3.2 *Let HTSP and HTSP1 be two typed specifications of σ and σ_1, respectively, and $\sigma \rightsquigarrow \sigma_1$. If both HTSP and HTSP1 can be implemented on SPEC, then there are enrichments EnSP and EnSP1 of SPEC, and implementations (HTSP, $impl$, EnSP) and (HTSP1, $impl_1$, EnSP1), such that diagram in Figure 1(b) commutes on K_{in}^{σ}, where h is an inheritance morphism.* ▌

4 Complex Classes and Their Inheritance

In this section we formalize the concept of complex class and inheritance on these classes in the framework of implementation studied in the last section.

A *complex class specification for a type HTSP* is an implementation CL=(HTSP, *imp*, SPEC) of HTSP by SPEC. A *model M* of a class CL=(HTSP,*impl*,SPEC) is a model of the implementation. An object of type σ in CL is a pair (a, b) for $a \in \sigma^A$ and b an object of ρ with $impl(a) = \bar{b}$, where $\bar{b}$ is the poset including b in the congruence. Given two typed specifications HTSP and HTSP1 of σ and σ_1, respectively, and two classes, CL=(HTSP, *impl*, SPEC) and CL1=(HTSP1, $impl_1$, SPEC1), we said that *CL1 is an inheritance extension of CL* if SPEC1 is an enrichment of SPEC; and if for each instance component $k \in K^\sigma$, $h \circ impl(k) \in impl_1(K^{\sigma_1})$, where h is the inclusion morphism from K^τ to K^{τ_1}, and for each meta-method α in K^σ, there is a meta-method β in K^{σ_1} such that $impl_1(\beta)$ is compatible with $impl(\alpha)$.

Proposition 4.1 *Let class CL1=(HTSP1,$impl_1$,SPEC1) be an inheritance extension of class CL =(HTSP,impl,SPEC). For a model M=(A,impl, B,$\equiv$) of CL1, let N=(A′,impl′, B′,$\equiv'$) be obtained from M by replacing σ^A by $\sigma^A \cup \sigma_1^A$ and replacing ρ^B by $\rho^B \cup \rho_1^B$, then N is model of CL1, where, ρ and ρ_1 are the central sorts in $SPEC_{impl}$ and $SPEC1_{impl_1}$, respectively.* ▮

Proposition 4.2 *With the conditions above, let N=(A′,impl′,B′,$\equiv'$) be obtained from M by replacing σ^A by σ_1^A and replacing ρ^B by ρ_1^B. If σ is strongly inherited by σ_1 in A, then N is a model of CL1 after compatibly adapting α to τ in A.* ▮

Similarly, these properties mean that objects of an inheriting class are objects of the inherited class.

References

[1] E. Astesiano, A. Giovini, G. Reggio, and E. Zucca. An integrated algebraic approach to the specification of data types, processes, and objects. In M. Wirsing and J. A. Bergstra, editors, *Algebraic Methods 1: Theory, Tools and Applications*, pages 91–116, 1989. LNCS 394.

[2] C. Beierle and A. Voβ. Implementation specifications. In H. Kreowski, editor, *Recent Trends in Data Type Specification: 3rd Workshop on Theory and Applications of Abstract Data Types*, pages 39–53, 1985. Informatik-Fachberichte 116.

[3] G. Blair, Gallagher, D. Hutchison, and Shepherd, editors. *Object-Oriented Languages, Systems and Applications*. Pitman, London, 1991.

[4] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*. Springer-Verlag, 1991. LNCS 562.

[5] T. S. Dillon and P. L. Tan. *Object Oriented Conceptual Modeling*. To be published by Prentice-Hall International, 1993.

[6] H. Ehrich, J. A. Goguen, and A. Sernadas. A categorical theory of objects as observed processes. In J. W. de Bakker, W. P. de Roever, and G. Rosenberg, editors, *Foundations of Object-Oriented Languages*, pages 203–228, 1991. LNCS 489.

[7] H. Ehrich and A. Sernadas. Algebraic implementation of objects over objects. In J. W. de Bakker, W. P. de Roever, and G. Rosenberg, editors, *Stepwisw Refinement of Distributed Systems, Models, Formalism, Correctness*, pages 203–228, 1989. LNCS 430.

[8] H. Ehrig, H. Kreowski, B. Mahr, and P. Padawits. Algebraic implementation of abstract data types. *TCS*, 20:209–263, 1982.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Mass., 1989.

[10] M. Groβe-Rhode. Towards object oriented algebraic specifications. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification, 7th workshop on specification of ADTs*, pages 98–116, 1991. LNCS 534.

[11] X.-M. Lu and T. S. Dillon. An algebraic theory for object oriented systems. 1991. Accepted for publication in *IEEE Transactions on Knowledge and Data Engineering*.

[12] X.-M. Lu and T. S. Dillon. Towards an algebraic theory of inheritance in object oriented programming. In preparation.

[13] B. Meyer. *Object Oriented Software Construction*. Prentice-Hall International, 1988.

[14] A. Poigné and J. Voss. On the implementation of abstract data types by programming language constructs. *JCSS*, 34(2/3):340–376, 1987.

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

An Object-Oriented Design for the ACT ONE Environment*

Martin Gogolla
TU Braunschweig
Informatik, Abt. Datenbanken
Postfach 3329
W-3300 Braunschweig
gogolla@idb.cs.tu-bs.de

Ingo Claßen
TU Berlin
Fachbereich 20
Franklinstr. 28/29
W-1000 Berlin 10
ingo@opal.cs.tu-berlin.de

1 Introduction

The overall aim of this paper is to stabilize and strengthen the algebraic specification method to software engineering and development. We do not introduce new theoretical results, but define a conceptual model, i.e., an information system schema, for the well-established algebraic specification language ACT ONE and its accompanying specification environment. Existing specification systems like ASSPEGIQUE [BC85], RAP [Hus85], ACT [Han87], OBJ3 [GW88] or OBSCURE [LW91] provide mechanisms to store and retrieve specifications and t^ operate on them. But in general (the OBSCURE system seems to be an exception) they employ no systematic approach to information administration. Usually they rely on storage facilities of the underlying programming language and file system. A solution to the information handling problem is the use of information systems or more specific databases. They are already accepted as being important components of software development systems, and, since specification systems can be regarded as parts of general software development systems, the same arguments apply to them.

2 Applying Database Technology to Algebraic Specifications

The application of databases to construct systems for specification languages leads to certain requirements: (1) Since specifications are structured entities, the database must be capable to deal with complex objects in a coherent way. This requirement suggests not to employ relational technology. (2) Since the specification task has an interactive nature, the system must be capable to deal with incomplete information.

*Work reported here has been partially supported by the CEC under Grant No. 6112 (COMPASS) and BMFT under Grant No. 01 IS 203 D (KorSo).

It must support different degrees of incompleteness and should enable mechanisms for automatic tool invocation if the state of completeness changes.

In our approach, we apply know-how of the database field in the area of algebraic specifications. The development of algebraic specifications describing software systems of practical relevance usually results in large sets of related specification units. These units arise from the decomposition of complex specifications into smaller pieces by means of the structuring mechanisms provided by specification languages. Additionally algebraic methods and especially specification languages give rise to a bulk of information like proofs, formal transformation steps, formal relations like signature morphisms, etc., which have to be stored to be accessible by various tools.

Here we show how a concrete data model, namely the object-oriented data model of TROLL *light* [CGH92], can be used to support the algebraic specification language ACT ONE. However, the concepts used are general enough to support other specification languages as well. Therefore, we feel the design of a conceptual schema for ACT ONE is mainly a case study in employing a semantic data model for database support of specification or programming languages. The approach chosen is general and can be used for other languages as well. It is therefore a proposal for the consolidation of environments for algebraic specification languages. The definition of the database schema is done by means of TROLL *light*, a specification language for objects developed recently within the KORSO Project. TROLL *light*, a dialect of TROLL [JSHS91], allows to represent structure and behavior of conceptual objects. It is designed to describe the universe of discourse as a system of concurrently existing and interacting objects. As in TROLL object descriptions are called templates in TROLL *light*. Because of their pure intensional meaning templates may be compared with the notion of class found in object-oriented programming languages. In the context of databases however, classes are also associated with class extensions so that we settled on a fresh designation. Templates show the following structure.

```
TEMPLATE name of the template
  DATA TYPES    data types used in current template
  TEMPLATES     other templates used in current template
  SUBOBJECTS    slots for sub-objects
  ATTRIBUTES    slots for attributes
  EVENTS        event generators
  CONSTRAINTS   restricting conditions on object states
  VALUATION     effect of event occurrences on attributes
  DERIVATION    rules for derived attributes
  INTERACTION   synchronization of events in different objects
  BEHAVIOR      description of object behavior by event-driven sequential machines
END TEMPLATE
```

3 ACT ONE Types Described by TROLL *light* Templates

We cannot go into the details of our design of the ACT ONE environment or into the details explaining how TROLL *light* can be translated to the object-oriented database system [LLOW91] used in the Braunschweig KORSO project. But in order to give a feeling how the design looks like we concentrate on ACT ONE types. An ACT ONE type is represented in TROLL *light* by a template (or object type) characterizing its static and dynamic properties.

The template **Type** given below has the following attributes: **Name** – Name of the type. **Text** – Textual representation of the type provided by some editor. **UsedNames** – List of used type names. **CfCorrect** – Indicates whether the textual representation has been checked syntactically, i.e., whether it is context free correct, and a syntax tree has been built. **Complete** – Indicates whether all types in the **UsedNames** list are in the database. **Flattable** – Indicates whether a flat representation of the type can be computed. **IsFlat** – Indicates that a flat representation is available. **UsedTypes** – Set of used and actually existing types. **Syntax** – Object-valued attribute describing the syntactical appearance of the corresponding type. This attribute may be undefined and will be defined after successful context free analysis. **Flat** – This attribute describes the flat representation of types. It may be undefined and will be defined after successful context sensitive analysis.

In contrast to the attribute **UsedNames** which contains a list of type names necessary for context sensitive analysis, but which may not be already existing, the set- and object-valued attribute **UsedTypes** refers only to those types which are currently existing.

```
TEMPLATE Type
  DATA TYPES   String, Bool;
  TEMPLATES    Type, Typeexpr, Pspec;
  ATTRIBUTES   Name:string;             Text:string;
               UsedNames:LIST(string);  CfCorrect:bool;
               DERIVED Complete:bool;   DERIVED Flattable:bool;
               IsFlat:bool;             UsedTypes:SET(type);
               Syntax:typeexpr;         Flat:pspec;
  EVENTS       BIRTH create(InitName:string,InitText:string);
                     changeText(NewText:string);
                     ...
               DEATH destroy;
  CONSTRAINTS  DEF(Name); DEF(Text);                     -- (R1)
               CfCorrect IMPLIES
                 (DEF(Syntax) AND DEF(UsedNames));       -- (R2)
  VALUATION    [create(N,T)] Name=N, Text=T;
               ...
  DERIVATION   Complete =
                 CfCorrect AND
                 (FORALL (N:LTS(UsedNames))
                   (EXISTS (T:UsedTypes)
                      (Name(T)=N AND CfCorrect(T)))); -- (R3)
               Flattable =
                 Complete AND
                 (FORALL (T:UsedTypes) IsFlat(T));       -- (R4)
  BEHAVIOR     ...
END TEMPLATE;
```

In the template certain requirements concerning ACT ONE types are formulated as constraints and derivation rules: (R1) A type must have at least a name and a textual representation. (R2) If a type has been checked syntactically its syntax tree and use list must be available. (R3) A type is complete if all used types are already existing. (R4) A type is flattable if all its used types are already flat. Please note that arbitrary events are possible in our approach. We could even have events like `contextFreeAnalysis`, `contextSensitiveAnalysis`, or `computeFlatRepresentation`.

4 Conclusion

Although our approach was inspired by [BCC90] and we tried to describe the same problems, our approach is quite different. In [BCC90] the design of the specification database of the ASSPEGIQUE environment is described by means of the algebraic specification language PLUSS. They employed a general specification language and presented a rather long specification describing certain states of incompleteness of specifications. Because we employ a powerful data model we are able to describe the same affair in fewer lines.

References

[BC85] M. Bidoit and C. Choppy. ASSPEGIQUE: An Integrated Environment for Algebraic Specifications. In H.Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'85)*, pages 246–260. Springer, LNCS 186, 1985.

[BCC90] M. Bidoit, F. Capy, and C. Choppy. The Design and Specification of the ASSPEGIQUE Database. In A. Miola, editor, *Proc. 1st Int. Symposium on Design and Implementation of Symbolic Computation Systems (DISCO'90)*, pages 205–214. Springer, LNCS 429, 1990.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL light: A Core Language for Specifying Objects. Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Research Report SRI-CSL-88-9, SRI International, 1988.

[Han87] H. Hansen. The ACT-System: Experiences and Future Enhancements. In D.T. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification (WADT'87)*, pages 113–130. Springer, LNCS 332, 1987.

[Hus85] H. Hussmann. Rapid Prototyping for Algebraic Specifications - RAP-System User's Manual. Technical Report MIP 8505, Computer Science Department, University of Passau, 1985.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91–04, Technische Universität Braunschweig, 1991.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreib. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[LW91] J. Loeckx and M. Wolf. The OBSCURE Manual. Technical Report 91/03, Computer Science Department, University of Saabrücken, 1991.

A Formal Definition of an Abstract Prolog Compiler

Julio García-Martín Juan José Moreno-Navarro
Universidad Politécnica de Madrid *

1 Motivation and related work

In the last years, the importance of logic programming languages has been increased. For logic languages we could understand not only PROLOG but also several languages that uses logical components (deductive inference as operational semantics, unification, backtracking, etc.). Probably, the development of efficient implementation techniques for PROLOG (the canonical element of these kind of languages) is an important component of their success.

The contribution of Warren [Wa83] with the design of an abstract machine points out the possibility of compiling PROLOG and getting efficient code. Most of current PROLOG systems are based on the resulting machine, usually called the WAM (Warren Abstract Machine).

Even though there are formal descriptions of the WAM (see [Ku89],Ru92 and, specially, [BR92]) the explanations (for instance [GLLO85], [MW88], and, best of all [AK91]) do not seem to approximate the reader to a good understandable view.

For our point of view, we think that it is possible and necessary to reinvent the WAM. This claim can be easily justified with the following words: a new and more clear view must be offered about compilation of PROLOG, but it must not be a collection of instructions being executed on an memory stack.

In this paper we present an abstract view of the WAM by a formal description. For an abstract WAM we understand a description of the WAM focused in how: a) it implements SLD-resolution with backtracking and b) the main elements of PROLOG (unification and backtracking) can be compiled. We are not interested in implementation details and optimisations.

The components of an abstract machine are the following:

- the *data area* which defines the *configuration* of the machine;
- the *instruction set* and a *semantic function* for each of its elements (defining the changes on the configuration after executing an instruction);
- the *transition function* between an initial and a final configuration which is guided by the semantic function of the instruction being currently executed; and
- the *translation function* which compiles a program into machine code.

Abstract data types (ADTs) can be used to describe these components, while the semantic function is defined in terms of the operations of the ADTs.

Furthermore, this definition is the middle point of a more ambitious project: The abstract WAM can be derived from SLD-resolution, the operational semantics of PROLOG, by stepwise refinement. Furthermore, the whole WAM can be derived from the abstract WAM by supplying efficient ADT implementations. Notice that the framework allows to manage both steps, by refining the data area (in the first step) or by refining ADTs implementation.

An executable and visualisable formal specification would point out the success of the design decisions taken by Warren in the compilation of PROLOG and could made them applicable to other logic languages.

2 The Abstract WAM

2.1 Data Area

This section informally describes the abstract WAM. We are using an OBJ-like language for the specification with some simple modifications in order to make it closer to the object oriented approach and to simplify the specification. For instance, if an ADT a is just the aggregation of some different ADTs a_1, ..., a_n (what is very often) we allow to use operations of a_i as operations of a without writing them in a's specification. We also allow the use of operations as arguments of other operations. Due to the lack of space we will only present some examples of the formalisation. Figure 1 shows the basic ADTs SET and STACK used later.

As shown in figure 2 the data area is formed by the WAM-program, or-stack, the argument registers and the heap. Let us discuss each element with some detail.

- The program contents a label-indexed array of WAM-instructions and a program counter, which is a label.

*Departamento LSIIS, Facultad de Informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain, email: {juliog,jjmoreno}@fi.upm.es.

```
adt SET [X : element] is
  sort set

    operation Insert : set element → set
    operation Remove : set element → set
    operation ∈ : element set → bool
    operation ∅ : → set
    operation _∪_ : set set → set

  axioms
    vars i, j : element
    var S : set
    vars A, B : set

    Insert (Insert (S,i),j) = Insert (Insert(S,j),i)        if i <> j
    Remove (Insert (S,i),j) = Remove (S,j)                  if i == j
    Remove (Insert (S,i),j) = Insert (Remove(S,j),i)        if i <> j
    Remove (∅,i) = ∅
    j ∈ Insert (S,i) = (i == j) or (j ∈ S)                  if (i <> j)
    ¬i ∈ ∅
    i ∈ (A ∪ B) = (A ∈ i) or (B ∈ i)
endadt
```

```
adt STACK [X : element] is
  sort stack

    operation EmptyStack : → stack
    operation Push : stack element → stack
    operation Pop : stack → stack
    operation IsEmpty : stack → bool
    operation Top : stack → element
    operation Mod_Top : stack (element→ element) → stack

  axioms
    var S : stack
    var e : element
    var f : element → element

    IsEmpty (EmptyStack()) = true
    IsEmpty (Push (S,e)) = false
    Pop (EmptyStack (S,e)) = error
    Pop (Push (S,e)) = S
    Top (Push (S,e)) = e
    Mod_Top (Push (S,e), f) = Push (S, f(e))
endadt
```

Figure 1: The ADTs SET and STACK

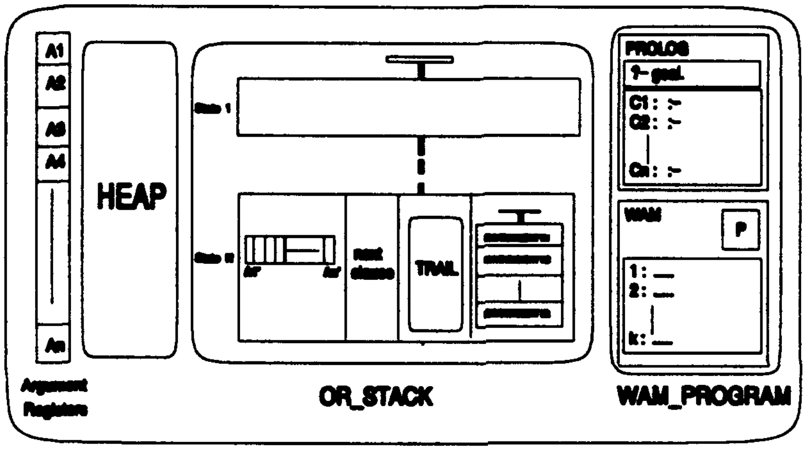


Figure 2: The Abstract WAM components

- The argument registers are collected in an array of heap pointers. It is used for parameter passing during clause application. The ADT argument registers is a simple instance of the basic ADT ARRAY.
- A stack (called or-stack) is used to traverse the resolution tree with a depth first strategy. The or-stack is defined by inheritance after instantiating the ADT stack with choice point as components.
- Choice points are used to store the information needed for applying any clause to a predicate call. There are several applicable clauses, so this information could be reused several times. The ADT choice point contents a copy of the argument registers (arguments of the predicate call), the local trail (used to record variable bindings in order to undo them after backtracking), the program address of the next clause and one and-stack.
- A trail is a set of variable names, as shown in figure 3.[1]
- As before, we get the and-stack after an instantiation and inheritance from STACK with environments as elements (see figure 3).
- An environment is used to perform the application of a given clause to a predicate call. For this purpose, it contains the continuation label (beginning of the code of the next predicate call) and the variables of the clause.
- The ADT variable is defined together with the ADT heap. A variable is a pair (variable name, heap pointer). The heap is used to represent PROLOG terms in clauses and goals. A heap is a table with a pointer as key and each element is a term: a constant, a constructor with some heap pointers as arguments or a variable name.

[1] Notice that the implementation of the trail as a stack is an optimisation detail.

```
adt TRAIL is
  extending SET [VAR_NAME] trail
endadt

adt ENVIRONMENT is
  sort environment
  uses VARIABLES WAM_PROGRAM

    operation Create : nat prog_addr → environment
    operation Modify_CP : environment prog_addr → environment
    operation Consult_CP : environment → prog_addr
    operation Modify_Var : environment nat variable → environment
    operation Consult_Var : environment nat → variable

  axioms
    vars size, n : nat
    var E : environment
    var CP : prog_addr
    var X : variable
```

```
    Consult_CP (Create (size, CP)) = CP
    Consult_CP (Modify_CP (E, CP)) = CP
    Consult_Var (Create (size, CP)) = Unbound
    Consult_Var (Modify_Var (E, n, X)) = X
endadt


adt AND_STACK is
  extending STACK [ENVIRONMENT] and_stack
endadt


adt OR_STACK is
  extending STACK [CHOICE] and_stack
endadt
```

Figure 3: The ADTs ENVIRONMENT, TRAIL, AND-STACK and OR-STACK

2.2 Semantic Function

We will not give the complete instruction set, but one can found (a part of) it in the next section where the translation function is discussed.

The semantic function is a mapping between a WAM-instruction and a data area into a data area, i.e. specifies the modification in the data area after executing a given instruction. The specification is the composition of several operations of the data area components. Figure 4 shows the specification of the semantic function of a couple of instructions. For instance, the try_me_else instruction rewinds the trail of the most recent choice point (by using an operation of the ADT or-stack), stores the label of the next alternative, initialises the choice point for the next clause application and increments the program counter.

```
  WAM_Program :=     goaltrans (GOAL)
                     proctrans (PROC (p1))
                        ...
                     proctrans (PROC (pn))
proctrans: Procedure → Wam_Code

proctrans (PROC (p) = {C1}) := clausetrans (C)

proctrans (PROC (p) = {C1,..., Cn}) :=

                     try_me_else (n, (label1))
                     clausetrans (C1)
         (label1):   retry_me_else (label2)
                     clausetrans (C2)
         (label2):      ...
       (labeln-1):   trust_me
                     clausetrans (Cn)


clausetrans: Clauses → Wam_Code

clausetrans (p(t) :- q1(s1),..., qn(sn)) :=
             allocate (#Variables (t, s1,..., sn))
             unifytrans (t)
             transfertrans (s1, var (t))
             call ((q1), arity (q1))
             transfertrans (s2, var (t, s1))
             call ((q2), arity (q2))
                ...
             transfertrans (sn, var (t, s1,..., sn-1))
             call ((qn), arity(qn))
             deallocate
```

```
SI: Wam_Instruction x Wam_State → Wam_State


SI [try_me_else (L)] wam_state :=
        Next_Instruction
            (Push (wam_state
                    Create_Choice-Point
                         (Registers (wam_state), (L),
                          Consult_CP_Program (wam_state))))

SI [retry_me_else (L) n] wam_state :=
        Next_Inst_Program
            (Rewind_Trail_Top_Or-Stack
                   (Next_Clause_Top_Or-Stack
                         (wam_state, (L))))


SI [allocate n] wam_state :=
        Next_Instruction
           (Mod_Top (wam_state
                        Push (Top (wam_state),
                              Create_Environment
                                 (n, Consult_CP (wam_state)))))


SI [proceed] wam_state :=
        Next_Instruction (Set_P (wam_state, Consult_CP (wam_state)))


SI [set_value Xn] wam_state :=
         Next_Instruction (Set_Val (wam_state
                                         Consult_Var (Top (Top (wam_state)),n)))
```

Figure 4: Translation and Semantic Function

2.3 Translation Function

The translation function specifies how PROLOG code is compiled into WAM code. It is described by using some auxiliary functions. Figure 4 describes the compilation of a PROLOG program as the compilation of the goal and the procedures (clauses for a given predicate). A procedure needs some code for the management of backtracking (try_me_else, retry_me_else and trust_me instructions) and the compilation of clauses. Clauses are translated by using the *unifytrans* and *transfertrans* schemes and particular instructions.

3 Conclusion

Although it is not the main goal of the paper, let us say something about our derivation of the WAM. It is carried out in two big steps. The first step is the *derivation of the main elements of the WAM*. We have not space enough to describe all the refinement steps. So, we would only mention some important points.

The preliminary machine is a stack based description of SLD-resolution solving literals from left to right and using the clauses in textual order. The stack stores resolution steps containing the current goal (a list of literal starting with a predicate call *p*), the substitution of the step and the next clause of p to be used. These resolution steps are called choice points.

Now, code could be used to codify goals. The goals in the choice points are replaced by some arguments registers and the continuation program label. A program label replaces the next clause.

Next step is the compilation of substitutions. The heap allows to represent substitutions as a set of pairs (variable name, heap pointer). The set includes the variables bound during the resolution step (choice point).

One could notice that the number of variables bound in a resolution step is unknown in advance. However, it is possible to give names to the local variables of the clause during the compilation process. The choice point could be responsible of collecting the bindings of the local variables. Nonlocal variable bindings are "remembered" into a local trail. From the point of view of SLD-resolution the trail helps in the reconstruction of the step substitution. From the machine viewpoint, it is needed to rewind variable bindings after backtracking.

Furthermore, in the case that a predicate has only one clause, a full choice point could suppose a waste of memory. It can be simplified in an environment with only local variables and the continuation label. The and-stack keeps all the environments belonging to a choice point.

As a final step, term representation into the heap and the parameter passing mechanism could be refined by using specialised machine instructions. These instructions have the responsibility of constructing or unifying terms (constant, functor or variable).

With this derivation we obtain the *Abstract WAM* described before. The result enhances the abstract behaviour of the WAM without knowing implementation details. The data area is configured with some abstract data types that are not fully implemented but the implementation must fulfil some axioms.

As a second big step, we can make the *optimisation* of this machine. The optimisations are performed in the same framework. Some optimisations arise from further refinement of the data area (for instance using a global trail instead of a local one). Other ones from the concrete implementation of the abstract data types: implementation of the heap as a stack, optimal memory allocation of the data area as contiguous memory areas, etc. Finally, the semantics of the instructions could also be optimised as in the last call optimisation, the environment trimming and so on. As a result we get a formal description of the WAM as described in [AK91].

The derivation and the Abstract WAM could help to understand the compilation of PROLOG. Moreover, they are useful to modify the machine design to implement new "logic languages" (in a general sense). The WAM have been used as a basis for the implementation of several declarative languages and symbolic computation systems: integration of functional and logic programming, constraint logic programming, logic programming with types, modules and contextual information, etc. The designer of a new machine could diverge from the WAM in any point of the derivation where the new language is different. The step by step specification has another advantage. The verification of the correctness of the WAM is simply obtained by proving equivalence between every machine and the following.

The Abstract WAM (or a similar abstract machine) is easy to implement and test. In this sense we also plan to make a computer visualisation of all the process. A first prototype [GM92] is ready and we expect to complete it soon.

References

[AK91] H. Aït-Kaci: The WAM: A (Real) Tutorial, The MIT Press, 1991

[BR92] E. Böerger, D. Rosenzweig: The WAM – Definition and Compiler Correctness, Technical Report TR 14/92, Dipartamento di Informatica, Università di Pisa, Italy, 1992.

[GLLO85] J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeck: A Tutorial for the WAM for Computational Logic, ANL-84-84, Argonne Nat. Lab., 1985

[GM92] J. García-Martín, J.J.Moreno-Navarro: Friendly-WAM An Interactive Tool to Understand the Compilation of PROLOG, Proc. LPAR 92, Springer LNCS 500, 1992

[Ku89] P. Kursawe: How to Invent a PROLOG Machine, New Generation Comp., 5, 1989.

[MW88] D. Maier, D.S. Warren: Computing with Logic: Logic Programming with PROLOG, Ed. Benjamin Cummings, 1988

[Ru92] D.M. Russinoff: A Verified PROLOG Compiler for the Warren Abstract Machine, J. of Logic Programming, 1992.

[Wa83] D.H.D. Warren: An Abstract PROLOG Instruction Set, Tec. Note 309, SRI International, Menlo Park, California, October 1983

Completeness of Equational Definitions over Predefined Algebras *

Valentin Antimirov[1] ** and Anatoli Degtyarev[2]

[1] Computer Science Department, Copenhagen University, 2100 Copenhagen, Denmark
email: anti@diku.dk; fax: (+45) 353-21401; tel.: (+45) 353-21400
[2] Department of Cybernetics, Kiev University, 252127, Kiev, Ukraine
email : caphedra%d105.icyb.kiev.ua

1 Introduction

The notion of *equational definitions over predefined algebras* (EDPA) was introduced in [3] in order to formalise the following rather widespread situation: given a data type D with a set of (predefined) functions Σ, a set of new (possibly *partial*) functions F on D is specified by a set R of "recursive equations" of the form

$$f(t_1, \ldots, t_n) = t \tag{1}$$

where $f \in F$ and $t_1, \ldots, t_n, t$ are terms over the signature $\Sigma + F$.

The construction covers a variety of known particular cases from both mathematics and computer science. First-order functional programs over predefined (built-in or "abstract") data types form a particular class of *functional* EDPA – in this case R is an F-indexed family of equations (1) where $t_1, \ldots, t_n$ is just a list of distinct variables. One can also recollect partial recursive definitions of arithmetic functions (over the algebra of natural numbers), term-rewriting systems over built-in algebras [1], or another EDPA of a more general form. E.g., the following two equational definitions over the algebra N of natural numbers with usual operations are intended to define (a) the greatest common divisor and (b) the integer division:

```
(a)  gcd(0,n) = n; gcd(m,n) = gcd(n,m); gcd(m+n,n) = gcd(m,n).
(b)  div(m,m+n+1) = 0 ;  div(m+n,n) = 1 + div(m,n).
```

Here gcd is presumably total, while div seems to be partial, but it is a matter of *semantics* to say precisely which functions on N are defined by these equations.

In [2,3] we have been developing *algebraic* semantics[3] of EDPA in order to make it possible to use equational logic with induction and corresponding term rewriting techniques for reasoning about functions defined in this way. A natural approach to this task is to represent the equational definition (1) as an enrichment (consistent, but not necessarily complete) of some algebraic specification SP of D. The main point here is to ensure that *any* correct specification of D provides *the same* semantics for a given set of equations R. To meet this natural requirement, we have introduced in [3] a flexible kind of algebraic presentations which leads to a so-called *safe* semantics of EDPA. In the next section we briefly reproduce this construction and then turn to the subject of *completeness* of EDPA.

* Short version

** On leave from V.M.Glushkov Institute of Cybernetics, Kiev, Ukraine

[3] *vs.* denotational one

2 Algebraic Presentations and Safe Semantics of EDPA

In what follows, $\mathcal{A}$ denotes a (predefined) Σ_0-algebra over an S-sorted signature Σ_0, $\Sigma_0 + F$ is a (signature) enrichment, and R is a set of $\Sigma_0 + F$-rewrite rules (oriented equations) of the form (1). Then the quadruple $(\Sigma_0, \mathcal{A}, F, R)$ (denoted also $(F, R)_{\mathcal{A}}$) forms an *equational definition (of F by R) over $\mathcal{A}$*.

To define semantics of EDPA means to set a correspondence between the quadruples and *basic interpretations* of F – sets of partial functions

$$F^{\mathcal{A}} = \{f^{\mathcal{A}} : \mathcal{A}_w \xrightarrow{\cdot} \mathcal{A}_s \mid f \in F_{w,s},\ w \in S^*,\ s \in S\} \tag{2}$$

which is to be in a proper logical relation to to the set of equations R.

To do this, we use algebraic specifications in a slightly generalized many-sorted language where the set X of variables used in axioms contains a distinguished subset X^+ of *safe* variables (then the variables in $X \backslash X^+$ are called *unsafe*). We write $E(X^+)$ and $SP(X^+)$ to reflect the fact that some of variables in the set of axioms E of the specification SP are safe.[4]

Definition 1. Let $SP(X^+) = (\Sigma, E(X^+))$ be an algebraic specification (called a *basic* one) of $\mathcal{A}$ in the sense that Σ is a finite enrichment (or extension) of Σ_0 and the Σ_0-reduct of the initial model $I(SP)$ is isomorphic to $\mathcal{A}$[5]. Then

- the enrichment $SP'(X^+) = SP(X^+) + (F, R)$ is called an *algebraic presentation (with safe variables)* of the EDPA $(F, R)_{\mathcal{A}}$;
- a $\Sigma + F$-substitution $\theta : X \rightarrow T_{\Sigma+F}(X)$ is called *safe* if $\theta(X^+) \subset T_{\Sigma}(X^+)$;
- a *restricted congruence* $=_{E:R}$ on the ground term algebra $T_{\Sigma+F}$ is the least one generated in a standard way by the set of equations $E(X^+) \cup R$ using only safe substitutions;
- the quotient $T_{SP'(X^+)} = T_{\Sigma+F}/_{=_{E:R}}$ is a *(standard) model* of $SP'(X^+)$;
- the presentation (enrichment) $SP'(X^+)$ is called
  - *safe-consistent* if the Σ-reduct of its standard model contains a subalgebra isomorphic to $I(SP)$ (i.e., to $\mathcal{A}$);
  - *safe-complete* if each congruence class $[t]_{E:R}$ of $T_{SP'(X^+)}$ contains some Σ-term;
  - *safe-persistent* if it is both safe-consistent and safe-complete.

Proposition 2 (cf. [3]). *If the presentation $SP'(X^+)$ is safe-consistent, then there exists the basic interpretation $F^{I(SP)}$ of F on $I(SP)$ (and so on $\mathcal{A}$) defined as follows for each $f \in F$:*

$$f^{I(SP)}(\ [t_1]_E, \dots\ [t_m]_E) = [\ f(t_1, \dots,\ t_m)\ ]_{E:R} \cap T_{\Sigma} \tag{3}$$

for all tuples $t_1, \dots, t_m$ of ground Σ-terms of appropriate sorts provided the right-hand side is not empty, otherwise $f^{I(SP)}$ is undefined on the arguments. Moreover, the enrichment of $I(SP)$ with $F^{I(SP)}$ forms a partial subalgebra of $T_{SP'(X^+)}$. □

[4] The presence of safe variables in E doesn't change standard algebraic semantics (and logic) of SP.

[5] For the sake of simplicity, we shall identify the predefined algebra $\mathcal{A}$ with the initial algebra $I(SP)$ – forgetting about a possible difference between their signatures.

An important problem coming from this construction is to characterise syntactically a class of basic specifications providing safe-consistent presentations (and so algebraic semantics) for any functional EDPA.[6] The following sufficient condition is a generalisation of our previous results on this topic.[7]

Theorem 3. *An algebraic presentation $SP(X^+) + (F, R)$ of the functional EDPA $(F, R)_{\mathcal{A}}$ is safe-consistent if each axiom $l = r$ of $SP(X^+)$ satisfies the following condition: any variable occurring non-linear in l or r is safe.* □

It is worth noting that presentations of this kind (*with safe non-linearity*) allow to use safely *inductive* equational theorems[8] of SP for proving theorems about new functions, because the basic interpretation (3) is consistent with all such equations valid in the predefined algebra $\mathcal{A}$.

Now we to turn to the safe-completeness property in order to investigate a class of (safe-consistent) EDPA defining *total* functions.

3 Safe Completeness and Persistency of EDPA

A *complete* EDPA is supposed to define a *total* basic interpretation $F^{\mathcal{A}}$ (i.e., consisting of total functions $f^{\mathcal{A}}$). Regarding algebraic presentations with safe variables, one can check that the basic interpretation defined by (3) is total iff $SP'(X^+)$ is safe-persistent. Combining this with Theorem 3, we obtain the following corollary for the class of presentations $SP'(X^+)$ with safe non-linearity of functional EDPA: the basic interpretation $F^{I(SP)}$ is total iff $SP'(X^+)$ is safe-complete.

To go further, one can vary the set of safe variables in $SP'(X^+)$ to obtain a spectrum of restricted congruences $=_{E:R}$, models $T_{SP'(X+)}$, and basic interpretations $F^{I(SP)}$. In the extreme case when $X^+ = \emptyset$, the presentation SP' becomes just a many-sorted enrichment and Def. 1 yields the usual "unrestricted" or "unsafe" notions of the least congruence $=_{E+R}$, consistency, completeness, and persistency. In general, $=_{E:R}$ is weaker than $=_{E+R}$, so consistency implies safe-consistency and safe-completeness implies completeness, but not vice versa. We have proved the following facts about the relations between these safe and unsafe properties.

Theorem 4. *If the presentation $SP'(X^+)$ is safe-persistent, then its unsafe version SP' (with $X^+ = \emptyset$) is persistent and defines the same (total) basic interpretation as uie first one. In particular case of functional EDPA, safe-completeness of the presentations with safe non-linearity implies persistency of SP'.* □

However, the coverse is not true:

Proposition 5. *There exists a functional EDPA and its (safe-consistent) presentation $SP'(X^+)$ with safe non-linearity such that the latter is not safe-complete, but becomes persistent when $X^+ = \emptyset$.* □

[6] because any functional euational definition admits well-defined denotational senmantics.

[7] cf. Theorems 10, 11 in [3].

[8] whose non-linear variables are also safe.

This means that sometimes the safe non-linearity requirement is still too strong and gives rise to a partial basic interpretation when it could be total – if all the variables were made unsafe. However, the following proposition demonstrates the opposite effect:

Proposition 6. *There exists a functional EDPA $(F, R)_A$ with a safe-consistent and not safe-complete presentation $SP'(X^+)$ such that its unsafe version SP' is complete and inconsistent (so can't provide any basic interpretation F^A).* □

To put another words, *junk can be the reason of confusion* – if one doesn't protect somehow basic axioms from it. The results of this paper show that the safe non-linearity condition is sufficient to provide such a protection for functional EDPA, but still is not always necessary. It is an interesting open problem to find a proper weakening of the condition which would hold any $SP'(X^+)$ safe-persistent whenever SP' is persistent.

4 Related Work

A simple and elegant approach to partial algebras within the usual framework of many-sorted (total) ones has been suggested in [4] in terms of *based specifications*. Our Def. 1 and Prop. 2 would give essentially the same semantics if we restricted ourselves with only unsafe presentations (with $X^+ = \emptyset$). But this would give rise to the problem with consistency pointed out in Prop. 6 (cf. also the "instructive example" in [3]).

Algebraic specifications with built-in algebras introduced in [1] are very similar to EDPA, but their semantics was defined through "completely protected" presentations $SP(X^+)$ with $X^+ = X$ (cf. also *stratified specifications* in [5]). This is another extreme case which captures only predefined algebras with *strict* operations and gives rise to certain problems with completeness (Prop.5). It would be interesting to try to extend the term rewriting theory presented in [1] to the more general class of presentations with safe non-linearity.

References

1. Avenhaus J., Becker K.: Conditional rewriting modulo a built-in algebra. Technical report (SEKI Report SR–92–11), 1992, 23p.
2. Antimirov V., Degtyarev A. Consistency of equational enrichments. In A. Voronkov, editor, *Logic Programming and Automated Reasoning. International Conference LPAR '92*. LNCS 624, pp. 393–402, Springer-Verlag, 1992.
3. Antimirov V., Degtyarev A. Semantics and consistency of equational definitions. In M. Rusinowitch, J.L.Rémy, eds. *Conditional Term Rewriting Systems, Third International Workshop, CTRS-92, Proceedings*. LNCS 656, pp. 67–81, Springer-Verlag, 1993.
4. Kreowski H.-J. Partial algebras flow from algebraic specifications. In *ICALP'87, Proc. Int. Coll. on Automata, Languages, and Programming*, LNCS 267, pp. 521–530, Springer-Verlag, 1987.
5. Smolka J., Nutt W., Goguen J., Meseguer J. Order-sorted equational computation. In H.Aït-Kaci and M.Nivat, editors, *Resolution of Equations in Algebraic Structures*, pp. 297–367, Academic Press, New-York, 1989.

An Algebraic Approach to Modeling in Object-Oriented Software Engineering

George J. Loegel*
Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, Michigan USA

1 Universal Algebras, Modeling and Software Engineering

Our research uses universal algebras in a model-based approach to the software engineering process. We organize the analysis, design and implementation of software systems by combining the paradigms of mathematical modeling and universal algebras. Models based on mathematical modeling principles and represented using universal algebras provide a practical alternative to both the common, *ad hoc* approaches to the software engineering process and other object-oriented methods. We have used universal algebra models to support the development phases of the software engineering process. Algebraic models unify many of the current object-oriented paradigms as well as defining another paradigm for object-oriented software engineering. Our results support using algebraic methods as a foundation for the software engineering process.

In this paper, we first describe the similarities between mathematical modeling and the software engineer's task, and then describe how to use these similarities to develop a software engineering process that starts with algebraic models of the real-world system. We define the software engineering process as the refinement of these models. We show how these universal algebra models are developed during the analysis phase, refined during the design phase, and used during the implementation phase of a software project. Our models are also used during the maintenance phase to provide design and implementation information. We show evidence that this paradigm for development is good and useful. Our work develops a general, algebraic model-based implementation technology. We believe these steps provide advantages for software engineering and define a viable alternative to present software engineering technology.

2 Mathematical Modeling and the Software Development Process

The fundamental principle underlying our work is the idea of a *model* in both the epistemological sense of Minsky [Min68] and Naur [Nau85a] and the *system* modeling sense of Zeigler [Zei76] and Casti [Cas89]. The purpose of a model is to represent information about a system. The model uses a formal notation to represent the information internalized by a programmer about the system. In our case, we use universal algebras as the formal notation for our model. We agree with Naur [Nau85a] that all systems are understood by programmers in terms of some internalized model but represented in some externalized, formal notation. As Naur points out in [Nau85b] and [Nau89], good notation encourages the internalization process The ability of a programmer to answer new questions about the model demonstrates that information has been internalized. We recast the software engineering problem as the development and transmission of algebraic models with their accompanying notation from one group to another.

An important advantage of employing algebraic models is the ability to use the theory of modeling

*Current Address: Superconducting Super Collider Laboratory, 2550 Beckleymeade Avenue, Dallas, TX 75237 USA email:loegel@sscvx1.ssc.gov as in Zeigler [Zei76] to develop and define terminology and use the theory of universal algebras as in [Meh90] to describe the development process. We use Zeigler's approach for model development as the starting point for our work. Zeigler defines the model building process as a series of fi·'e steps:

1. identify the components;
2. identify the interactions between the components;
3. simplify the model;
4. build a computer simulation of the model; and
5. validate the model

Zeigler's approach, by focusing on the objects visible to the modeler, embodies the fundamental ideas of object-oriented software engineering. Further, the use of universal algebras to represent the models provides us with a notation that is both concise and flexible enough to describe various software systems. Even computer languages like SIMULA67 [Dah72], developed for modeling, use the concepts of universal algebras to describe abstract data types as defined by the ADJ Group [Gou78] and Zilles [Zil80]. The relationship between the theory of modeling and software engineering allows us to unify many of the model-based object-oriented software engineering approaches.

3 Software Engineering with Universal Algebras

We now describe the steps in our algebraic software engineering process and then apply these steps to developing a software system. We relate the steps in our process to the steps in the mathematical modeling process and show how we can use the interpretation of universal models to describe the process at each step.

Our initial or analysis model uses the customer's description of the components to produce a system specification using a universal algebra. Tse's dissertation [Tse91] shows how we can use a diagram to communicate with ·'.e customer and represent all of the information in a universal algebra. The design phase refines the analysis model by introducing new objects and using the resources available to determine the concrete data structures and algorithms. This refinement is a homomorphic transformation of the analysis model. The implementation phase converts the data structures and algorithms in our design model into statements in a programming language. Our entire process can be characterized in terms of universal algebra models and the universal algebra gives us a uniform notation for each step in the process.

4 Case Studies

We have used this algebraic approach in several systems. The first, described in the 1984 POPL [Mil84], used an algebraic description of attribute grammars to generate Pcode from Pascal. We produced a more compact and understandable description of the Pascal-to-Pcode translation than the corresponding compiler from ETH Zurich [Nor76]. Although the underlying system, Paulson's Compiler Generator (PCG) [Pau82a] limited us to a fixed set of primitives for building the algebra, we were able to define domains and operations on those domains. Also, PCG was a *declarative* system in that we only specified local rules and PCG determined the sequence for applying those rules. This project showed a means of prototyping a language using a direct implementation of the algebraically described semantics.

A second system, the Capture Storage Element (CSE) of the Optical Digital Image Storage System (ODISS), showed how we used an algebraic model to develop a system originally specified using another notation. ODISS also shows how the objects seen in the system by the customer are beneficial during development. That is, the software should reflect the way in which the customer perceives the tasks. ODISS is a distributed document storage system originally specified using a data flow diagrams. The CSE provided intermediate storage for documents before they were written to optical disk. ODISS was developed by Systems Development Corporation[1] to digitize and store Civil War documents for the National Archives and Records Administration (NARA) of the United States.

The algebraic model developed for the CSE was based on *documents*, unlike the other subsystems in ODISS which were based on *pages*. The algebraic description provided the basis for the user documentation and the implementation (≈ 17,000 lines of C). During the fifteen months of development and inte-

[1] Now Paramax, a subsidiary of Unisys

gration, only one integration error occurred due to misunderstanding the notation and only one serious error was found after delivery. Further, being able to examine the state of documents became a major tool during the integration phase of the project. After delivery, one of the first requests from the user was the ability to query document status, and this capability was easily added.

Since C is not object-oriented, the algebraic description became a key reference document during the development and permitted a ready assessment of the state of the implementation. Our experience with ODISS shows how an model-based algebraic design, derived from another notation, for defining the interface and guiding the implementation of a software system.

The third system we developed was a code optimizer for a portable compiler, where we demonstrated how modeling produced a working system faster than other approaches. This work was done as part of an advanced course in compiler construction. The class divided into three teams. One team started with Peter Bird's CoGG system [Bir82], another team used a simple parser-based technique, and we used modeling and simulation. Each team started with the portable BCPL compiler [Ric80] which had recently been ported to a Motorola 68000 system using a simple version of the macro expansion technique described in Strachey's GPM [Str65].

The BCPL complier produces an intermediate code (called OCODE) for a stack-based virtual machine. The intermediate code changes the code generation problem from one of mapping a high-level language to machine code into mapping a low-level intermediate code to machine code. One technique for code generation particularly suited for mapping OCODE to a target machine is simulation. We used the simple code generator as the starting point for our code generation model and used a universal algebra to describe the simulation process. We used different signatures for the universal algebra to define different optimizations. The implementations differed in the amount of state information carried in objects in the system. Out of the three teams, each of which started with a working compiler, we were the only ones to have a working compiler at the end of the course. Our exploitation of the original code generation model by expanding its simple signature played an important part in our success.

Our current work-in-progress is the Global Accelerator Control System (GACS) for the Superconducting Super Collider (SSC) Laboratory, a high-energy physics project being built near Dallas, Texas. The SSC will be the largest scientific instrument ever built. The proposed design for the control system has much in common with our algebraic models. The GACS will be based on EPICS[2], a control system designed at Los Alamos National Laboratory. EPICS has many of the features present in our other models. For example, the primitive objects in EPICS are classified based on the type of signal processed (binary, analog) and the update frequency. This means that physicists using EPICS do not need expertise in writing device drivers or working with real-time kernels. The physicists sees a model of the accelerator described in terms familiar to the physicists. This is analogous to ODISS where an archivist sees a system that organizes pages as documents which is the same way the archivist organizes pages.

EPICS currently provides a control system for small accelerators throughout the United States. Just as in the BCPL optimizer project, the accelerator model provided by EPICS must become more sophisticated to support the additional complexities of the SSC. We have proposed the same kind of algebraic modeling approach used in the BCPL optimizer project as a viable means to expand the capabilities of EPICS to meet the requirements of the SSC.

These systems show four applications of algebraic software engineering, all of which started with a model of the application described using a universal algebra. Each of these systems used algebraic descriptions to develop the design and implementation,and performed well with respect to various measures.

References

[Bir82] Bird, P. L. "An Implementation of a Code Generator Specification Language for Table Driven Code Generators", in Proceedings SIGPLAN82 Symposium on Compiler Construction, ACM SIGPLAN Notices, v. 17, no. 6, 1982, pp. 44-50

[Cas89] Casti, J. *Alternate Realities: Mathematical Models of Nature and Man*, Wiley-Interscience, 1989

[Dah72] Dahl, O.-J. and Hoare, C. A. R. "Hierarchical Program Structures", in *Structured Pro-*

[2] Experimental Physics and Industrial Control System

gramming, Dahl, O.-J., Dijkstra, E. W. and Hoare, C. A. R., Academic Press, 1972

[Gou78] Gougen, J. A. Thatcher, J. W. and Wagner, E. G. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types" in *Current Trends in Programming Methodology IV: Data Structuring*, Yeh, R. (ed.), Prentice-Hall, pp. 80-144, 1978

[Meh90] Mehlhorn, K. and Tsakaldis, A. "Data Structures" in *Handbook of Theoretical Computer Science Volume A: Algorithms and Complexity* (J. van Leeuwen, Editor), Elsevier Science Publishers B.V., pp. 300-341, 1990

[Mil84] Milos, D. Pleban, U. and Loegel, G. "Direct Implementation of compiler specifications, or: The Pascal P-compiler revisited", Conference Record of the 11th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, 1984, pp. 196-207

[Min68] Minsky, M. "Matter, mind and models", in *Semantic Information Processing*, M.I.T. Press, 1968

[Nau85a] Naur, P. "Programming as Theory Building", Microprocessing and Microprogramming, v. 15, 1985, pp. 253-261 (also in [Nau92])

[Nau85b] Naur, P. "Intuition in Software Development" in *Formal Methods and Software Development*, v. 2, Ehrig, H. Floyd, C. Nivat, M and Thatcher, J. (eds.), Lecture Notes in Computer Science 186, pp. 60-79 (also in [Nau92])

[Nau89] Naur, P. "The Place of Strictly Defined Notation in Human Insight", in *Proceedings of the Workshop on Programming Logic*, Dybjer, P. Hallnäs, L. Nordström, B. Petersson, K. and Smith, M. J. (eds.) Report 54, Programming Methodology Group, University of Gōtenborg, May 1989, pp. 429-423

[Nau92] Naur, P. *Computing: A Human Activity*, Addison-Wesley, 1992

[Nor76] Nori, K. V., Ammann, U., Jensen, K., Nageli, H. H., Jacobi, C. *The Pascal (P)-Compiler : Implementation Notes* (Revised Edition), ETH Zurich, Institut fur Informatik, 1976

[Pau82a] Paulson, L. *A Compiler Generator for Semantic Grammars*, Ph.D. dissertation. Stanford University, 1982

[Ric80] Richards, M. and Whitby-Strevens, C. *BCPL - The language and its compiler*, Cambridge University Press, 1980

[Str65] Strachey, C. "A general purpose macrogenerator", The Computer Journal, V. 8 1965, pp. 225-241

[Tse91] Tse, T. H. *A Unifying Framework for Structured Analysis and Design Models: An Approach using Initial Algebra Semantics and Category Theory*, Cambridge University Press, 1991

[Van89] Van Horebeck, I. and Lewi, J. *Algebraic Specification in Software Engineering : An Introduction*, Springer-Verlag, 1989

[Wei76] Weisenbaum, J. *Computer Power and Human Reason: From Judgement to Calculation*, W. H. Freeman, 1976

[Zei76] Zeigler, B. P. *Theory of Modelling and Simulation*, Wiley-Interscience, 1976

[Zei90] Zeigler, B. P. *Object-Oriented Simulation with Hierarchical, Modular Models; Intelligent Agents and Endomorphic Systems*, Academic Press, 1990

[Zil80] Zilles, S. N. "An Introduction to Data Algebras", in *Abstract Software Specifications*, Goos, G. and Hartmanis, J. (eds.), Lecture Notes in Computer Science no. 86, Springer-Verlag, 1980

An Automated Proof of the Correctness of a Compiling Specification

E.A. Scott, Mathematics and Computational Sciences, University of Surrey, U.K.

In this paper we discuss an automated proof of the correctness of a compiler. The source language for the compiler is PL_0 [8], a subset of OCCAM2 [5]. The target language, ML_0, is based on the machine language for the transputer [6]. Since the early work of Cohn [2] in the LCF system, compiler proofs have attracted a lot of attention as test cases for automated theorem provers, see for example [11] and [12]. Recently Broy [1] has used the Larch Theorem Prover to verify a code generator for a functional language. Our work differs from earlier studies in that we start with a detailed hand proof of compiler correctness and attempt to use a theorem prover to verify the proof.

The Languages PL_0 And ML_0

Intuitively, we expect to call a compiler correct if for all programs p, p and its compiled version have the same meaning. However, to give any kind of formal proof we must first formally define the semantics of the source and target languages. We use the approach that was developed in [9] and [7]. The basic idea is to begin by defining an extension PL_0^+ of PL_0. The syntax of PL_0^+ is given in standard BNF fashion. A refinement relation $\sqsubseteq$ is defined on PL_0^+ which captures enough of the semantics of the language to prove the results. Since PL_0 is a subset of PL_0^+ its semantics are inherited directly. The key aspect of this approach to compiler correctness, which was developed in [4], is that the necessary properties of the semantics of ML_0 are also *defined* in terms of PL_0^+. There is given a function I from ML_0 to PL_0^+, and the meaning of process m in the language is defined to be the meaning of $I(m)$ in PL_0^+. This allows a direct comparison of the meanings of elements of PL_0 and ML_0.

The function I is the composition of two functions *mtrans* and *Interp*. The function *mtrans* takes ML_0 instructions and translates them into transputer code. The function *Interp* takes lists of transputer code and returns PL_0^+ processes.

$$\begin{tikzcd}
PL_0 \arrow[r, "C"] \arrow[d, "identity"'] & ML_0 \arrow[r, "mtrans"] & \text{transputer code} \arrow[dll, "Interp"] \\
PL_0^+ & &
\end{tikzcd}$$

The main objection to this approach states that the semantics of a machine language cannot be *defined* in this way because there will be a prescribed semantics given naturally by the induced machine behaviour. In [7] this issue is not addressed, it is assumed that the semantics are defined by PL_0^+. If we were to begin with prescribed semantics for ML_0 it would be necessary to prove the properties which in this work are defined by the function I, i.e. we would have to prove the correctness of I. This should be possible provided that the prescribed semantics are sufficiently explicit, for the properties assumed in this work are all explicitly stated in the LP specification of PL_0^+. An alternative approach is to consider the interpretation I as providing a specification for the target language. Then we have (partial) specifications for source languages, target languages and compilers together with a proof that the compiling specification is correct for all languages satisfying the language specifications. As the aim of our work is to study the automation of the proofs given in [7], we shall take this view.

An advantage of the refinement relation approach is that proofs carried out are valid for any language which has the properties described by $\sqsubseteq$. Thus if PL_0, and hence PL_0^+, are later extended to richer languages the proofs discussed in this work will remain valid provided the properties required for the proofs still hold. Thus it is important that all the properties used in the proofs are explicitly stated so that it is clear what must be preserved in future extensions.

Compiler Correctness

For a given compiler C we cannot expect to be able to prove that p and $I(C(p))$ are equal. The compiled version of a program will contain identifiers, corresponding to things such as the program pointer and error flag, for which there will be no analogous identifiers in the original program. Thus we have to consider a PL_0^+ process Q_p that renames the identifiers in $I(C(p))$ and ends the scope of those identifiers introduced for machine purposes. It is reasonable to assert that $\mathrm{SEQ}(Q_p, p)$ has the same meaning as p, where SEQ is concatenation of PL_0^+ processes. Thus we formally define a compiler C to be correct if, for all PL_0 processes p, we have that

$$\mathrm{SEQ}(Q_p, p) \sqsubseteq \mathrm{SEQ}(I(C(p)), Q_p).$$

In [7] there are given sets of conditions $\mathcal{C}_p$ on ML_0 programs, and theorems of the form

$$\text{If} \quad m \quad \text{satisfies} \quad \mathcal{C}_p \quad \text{then} \quad \mathrm{SEQ}(Q_p, p) \sqsubseteq \mathrm{SEQ}(I(m), Q_p).$$

The theorems show that for a correct compiler C it is sufficient to take $C(p)$ be any sequence of code m which satisfies $\mathcal{C}_p$. Thus the set of all the $\mathcal{C}_p$ can be thought of as a compiling specification and the theorems prove that this specification is correct. These theorems are proved by hand in [7]. This work is an attempt to give automated proofs. The theorem prover used is the Larch Prover (LP) [3].

Automating The Proofs

When automating an existing hand proof there are two aspects to be considered:

(i) Can the system in which the proof is to be carried out be specified in the logic of the theorem prover?

(ii) Are the proof techniques of the theorem prover able to prove the results?

In this study (i) is equivalent to 'can we specify PL_0^+ in the logic of LP?' Answering this question turned out to be a major project in its own right, see [10]. In this paper we concentrate on (ii), using LP to prove the theorems within the specification of PL_0^+ which was developed in [10].

It is our experience that if a system can be specified in the logic of LP but an original hand proof cannot be reproduced using LP then this is because the original proof contained mistakes. There are two kinds of mistakes: those that can be corrected and those that cannot. A mistake is *correctable* if there exits a correct proof the result and *uncorrectable* if the result can not be proved. In the case of correctable mistakes we have been able to find a correct proof using the theorem prover. Uncorrectable mistakes can arise in two ways: either a misunderstanding of an implicit assumption led to the mistaken belief that a result should be true, or the original specification does not have the properties that were intended. In the first case once the misunderstandings were identified we were able to produce revised, provable versions of the results. In the second case the specification was modified to allow the proof of the results. Such modifications usually involved 'tightening up' implicit assumptions.

We have automated proofs of the correctness theorems for SKIP, STOP, assignment and the operator SEQ, together with the correctness of expression compilation for identifiers, integers and sums of expressions. We have not proved all the correctness theorems, they are not all proved in [7], however we have proved a sufficiently wide range to show that all the theorems could be proved by LP if the effort were considered to be worthwhile.

The Larch Theorem Prover

The Larch Prover is an equational reasoning theorem prover developed at MIT by S. Garland and J. Guttag [3]. It is intended primarily as an interactive proof assistant or debugger, and it is in this capacity that we have used it. LP is a theorem prover for a subset of multisorted first-order logic with equality. Equations are asserted by the user then ordered by LP into a rewrite system which can be used to prove other equations. The logic also contains *deduction rules*, statements of the form

When [(FORALL $x_1, \ldots, x_n$)] ⟨hypotheses⟩ Yield ⟨conclusions⟩

where x_i are variables, and where ⟨hypotheses⟩ and ⟨conclusions⟩ are sequences of equations. A specification in the LP logic can be axiomatized with induction rules. The statement

assert *sort* generated by *operators*

ensures that the only elements of *sort* are those that can be constructed using the specified *operators*. Results are proved by term rewriting; the rules are used to simplify both sides of an equation until a known equality is obtained. LP also supports proofs by induction, cases, and contradiction, and equations can be proved by performing critical pair calculations. See [3] for a full description of LP.

Results

As a consequence of the attempt to automate the proofs we discovered both correctable and uncorrectable mistakes. In the case of correctable mistakes the proofs were easily modified and we only mention these in passing. The discovery of uncorrectable mistakes lead to the need to modify both the specification of PL_0^+ and the formal definition of complier correctness to allow the results to be proved.

Modifications to the specification of PL_0^+ were necessary because there were not enough laws given in the original specification to prove the theorems. In particular we have had to add extra properties to the specification of identifiers and assignment, and we have had to give a more precise definition of the function *Interp*. The addition of extra properties is not a serious problem because the original specification was never intended to be complete. Rather it was just meant to be detailed enough to allow the proofs of the theorems, see [7]. So we merely added the necessary extra laws to the specification. The problems with the definition of *Interp* were correctable errors in the above sense. Essentially all that was involved was the addition of some assignments which ensured that the proofs followed from the specific laws stated and did not rely on any implicit assumptions.

A more serious problem was that the definition of the correctness of expression compilation given in [7] could never be satisfied by any compiler. This is an example of a mistake where the incorrectness of the result was unnoticed because some assumptions about the original specification were only made implicitly. Once these assumptions were identified we were able to reformulate the definition of correctness so that the result was true.

We also found that the theorems in [7] were not sufficient to prove that $C(p)$ would be correct for all p. The argument that the theorems prove the correctness is an inductive one: $C(p)$ is proved to be correct for all basic processes p, and then complex processes are dealt with under the assumption that all subprocess are known to be correct. For example, $C(SEQ(p,q))$ is proved correct under the assumption that $C(p)$ and $C(q)$ are known to be correct. However, when $r = \text{SEQ}(p,q)$, we need

$$\text{SEQ}(Q_r, p) \sqsubseteq \text{SEQ}(I(m), Q_r)$$

to prove that $C(r) = m$ is correct. Thus we needed to prove stronger theorems of the form:

If m satisfies C_p then $\text{SEQ}(Q_r, p) \sqsubseteq \text{SEQ}(I(m), Q_r)$, for any process r that has p as a subprocess.

After correcting these and other minor errors, we were able to use LP to produce automated proofs of the specification theorems.

The pragmatic conclusions that can be drawn from this work are that the (modified) compiler specification is correct, and that there already exist automated theorem provers capable of showing this. Furthermore, the compiling specification was developed independently of the automation, so this is a good test of the capabilities of the theorem prover used. However, perhaps the most powerful conclusion to be drawn from this study is the importance of automated theorem provers in the detection of mistakes in implicit aspects of a hand proof. It is in the implicit assumptions of a hand proof that errors most often occur and remain undetected (by human checkers). Automated proofs require implicit aspects to be made explicit thus exposing such errors.

References

1. M.Broy, *Experiences with machine supported software and system specifications and verification using LP, the Larch proof assistant*, preprint, October 1992.

2. A.Cohn, *Machine assisted proofs of recursion implementation*, Ph.D. Thesis, Dept. of Comp. Sci., University of Edinburgh, 1979.

3. S.J.Garland, J.V.Guttag, *An overview of LP, the Larch Prover*, In: N. Dershowitz, ed, Proc. 3rd International Conf. Rewriting Techniques And Applications, Lecture Notes In Computing Science **355** 137–151, Springer–Verlag, 1989.

4. C.A.R. Hoare, He Jifeng, *Refinement algebra proves correctness of compilation*, preprint, 1990.

5. INMOS Ltd, *Occam 2 reference manual*, Series In Computing Science, Prentice-Hall, 1988.

6. INMOS Ltd, *Transputer instruction set: a compiler writers guide*, Prentice-Hall, 1988.

7. He Jifeng, P. Pandya, J. Bowen, *Compiling specification for ProCos level 0 language*, Procos Technical Report [OU HJF 4], 1990.

8. H.H. Løvengreen, K.M. Jensen, *Definition of the ProCoS programming language level 0*, Procos Technical Report [ID/DTH HHl 2], 1989.

9. A.W. Roscoe, C.A.R. Hoare, *The laws of occam programming, Theoretical Computer Science* **60**, 177–229, 1988.

10. E.A. Scott, K.J.Norrie, *A study of PL_0^+ using the Larch Prover*, to appear in: Proceedings of the 1st International Workshop on Larch, Workshops in Computer Science Series, Springer, 1993.

11. D. Weber-Wulff, *Proof movie, Proving the Add-Assign Compiler with the Boyer-Moore Prover*, to appear in: Formal Aspects Of Computing.

12. W.D.Young, *A mechanically verified code generator*, Journal of Automated Reasoning, 5, 1989.

System Demonstrations

AMAST'93

Third International Conference on Algebraic Methodology and Software Technology

University of Twente
The Netherlands

Participants' Proceedings

RELVIEW – A Computer System For the Manipulation of Relations

Rudolf Berghammer and Gunther Schmidt
Fakultät für Informatik, Universität der Bundeswehr München
Werner–Heisenberg–Weg 39, D–85577 Neubiberg

People working with relations (e.g., in the theory of partial orderings, lattice theory, or graph theory) very often a use greater or smaller example and manipulate it with pencil and paper in order to prove or disprove some property. For supporting such a task by machine (and also since manipulation by hand is no more feasible with bigger examples), the RELVIEW system ([Berghammer Schmidt 91]) has been constructed at the Bundeswehr-University at Munich. The system is written in C and is currently available for Sun workstations with American National Standard C and Sunview 4.0.

RELVIEW is a totally interactive and completely video-oriented computer system for the manipulation of concrete relations which are considered as Boolean matrices. Its screen is divided into two parts. The left part is the drawing-window; here matrices can be drawn and manipulated using a mouse. The right part contains the command buttons and the scrollbars. The scrollbars can be used for showing a part of a relation the size of which exceeds the maximal window size. Also textual input (e.g., dimensions or names of relations) and output (e.g., results of tests, error messages) is requested and shown, respectively, in this part.

One relation, the so-called working copy, is displayed on the screen for editing. A whole collection of relations can be kept in the working memory during a working session. Such a collection may also be saved on permanent memory, e.g., on a hard disk. If a stored relation from the memory is displayed into the drawing-window for editing, a duplicate working copy is created. Editing with the mouse does only affect the working copy and thus does not change the original. To overwrite the original by the working copy, a specific RELVIEW command has to be used.

Execution of system commands is possible by clicking on command buttons. If a command requires arguments, then execution starts not before the last argument is given. Thus, if the user inadvertently has chosen a wrong button, undo consists in choosing the correct button – provided the argument input has not been finished. Besides some management commands, first, the system provides commands implementing the basic operations on relations. Furthermore, we have commands for residuals, quotients, and closures, for certain tests on relations, and commands which implement the operations important in relation-algebraic domain description (compare [Berghammer et al. 89, Zierer 91]). And, finally, RELVIEW allows the user to define and apply its own functionals on relations, where in the case of a unary functional with identical domain and range also repeated application is possible. A useful fact in applications is that the latter command can be used to compute fixpoint of monotone functionals. For instance, if the homogeneous relation R is contained in the working memory and one declares a RELVIEW functional

```
initial        -(R * -%),
```

where % stands for the variable, * means multiplication, and – means negation, then a repeated application of this functional to the empty vector yields the vector of the points from which only paths of finite length emerge. (Compare the definition of the initial part of a graph in [Schmidt Ströhlein 89], Section 6.3.)

A detailed description of how to draw on the drawing-window, how to use the scrollbars, and how to execute a command (inclusive parameter passing and result delivery) is given in [Abold-Thalmann et al. 89] and [Berghammer 92]. The first report also presents some implementation details, e.g., the internal representation of relations, and outlines fast algorithms for computing products, symmetric quotients, and residuals of relations. In the second report, also an example for prototyping using RELVIEW is presented, viz. the computation of the cut completion of a partially ordered set.

In the meantime, a lot of other studies have been performed with the RELVIEW system including further graph- and order-theoretic questions resp. algorithms, DAG-languages, domain constructions, relational specifications, and relational semantics. Of course, computation with RELVIEW is limited in space and time. The limit, however, depends heavily on the type of problem handled. As an example, we mention again the computation of the initial part. On our installation (SUN SPARCstation 10), we have treated, e.g., graphs with up to 5000 points.

Let us close with a few remarks on further developments on RELVIEW. It turns out that the system is a good tool for the interactive manipulation of relations. However, experience has shown that for some tasks certain additional features will be very helpful. A main improvement is possible in the layout. The present Boolean matrix visualization of relations is well-suited for many tasks, in particular, if the intention is to get insight into an "abstract" relational problem. However, if the system is used to solve concrete problems on graphs or related structures by relational methods, then it seems better to visualize homogeneous relations as directed graphs. Therefore, for the future we plan the incorporation of commands realizing a transition between Boolean matrices and graphs. Especially, it should be possible to edit a relation as a graph. For a visualization of results, furthermore, the user should be given the option to display a relation on the screen as a directed graph and to emphasize a specific subset of the nodes described by a vector.

Besides this main extension, we plan also some minor extensions of RELVIEW. E.g., we are concerned with interfaces to other systems. The ability for producing scientific papers on relations which mix text and drawings of Boolean matrices and graphs, respectively, can be obtained by interfacing the RELVIEW system with some typesetting systems. Furthermore, an interface to the relational formula manipulation system and proof checker RALF (also developed at Bundeswehr-University Munich [Brethauer 91]) is planned.

References

[Abold-Thalmann et al. 89] Abold-Thalmann H., Berghammer R., Schmidt G.: Manipulation of concrete relations: The RELVIEW-system. Report Nr. 8905, Fakultät für Informatik, Universität der Bundeswehr München (1989)

[Berghammer 92] Berghammer R.: Computing the cut completion of a partially ordered set – An example for the use of the RELVIEW-system. Report Nr. 9205, Fakultät für Informatik, Universität der Bundeswehr München (1992)

[Berghammer Schmidt 91] Berghammer, R., Schmidt, G.: The RELVIEW-system. In: Choffrut C., Jantzen M. (eds.): Proc. STACS '91, LNCS 480, Springer, 535–536 (1991)

[Berghammer et al. 89] Berghammer R., Schmidt G., Zierer H.: Symmetric quotients and domain constructions. Inform. Proc. Letters 33, 3, 163–168 (1989/90)

[Brethauer 91] Brethauer R.: Ein Formelmanipulationssystem zur computergestützten Beweisführung in der Relationenalgebra. Diplomarbeit, Fakultät für Informatik, Universität der Bundeswehr München (1991)

[Schmidt Ströhlein 89] Schmidt G., Ströhlein T.: Relationen und Graphen. Springer (1989); English version: Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Comput. Sci., Springer (1993)

[Zierer 91] Zierer H.: Relation algebraic domain constructions. Theoret. Comput. Sci. 87, 163–188 (1991)

Towards an Integrated Environment for Concurrent programs Development
(Proposal for a Demonstration)

Naïma BROWN and Dominique MERY*
CRIN-CNRS & INRIA Lorraine, BP 239
54506 Vandœuvre-lès-Nancy, France.
FAX: 33 83 41 30 79.
email: brown@loria.fr, mery@loria.fr

Formal methods comprise two aspects, namely *formal specification* and *verified design*. The methodology underlying these methods is first to specify precisely the behaviour of a piece of software, then to write this software and finally to prove whether or not that actual implementation meets its specification. This final aspect of formal methods is known as *verified design*[1]. *Unity* [CM88, M92, Kna90], as the *action systems* approach [BS91], is a formal method that attempts to decouple a program from its implementation. Therefore, *Unity* separates logical behaviour from implementation, provides predicates for specifications, and proof rules to derive specifications directly from the program text. This type of proof strategy is often clearer and more succinct than arguing about a program's operational behaviour.

Our research fits into *Unity*'s methodology. Its aim is to develop a proof environment suitable for mechanical proof of concurrent programs [BM93]. This proof is based on *Unity* [CM88], and may be used to specify and verify both safety and liveness properties. Our verification method is based on theorem proving, so that an axiomatization of the operational semantics is needed. We use Dijkstra's *wp*-calculus to formalise the *Unity* logic, so we can always derive a sound relationship between the operational semantics of a given *Unity* specification and the axiomatic one from which theorems in our logic will be derived. In a mechanically verified proof, all proof steps are validated by a computer program called a *theorem prover*. Hence, whether a mechanically verified proof is correct is really a question of whether the theorem prover is sound. The theorem prover used in our research is *B-Tool* [CL91c, CL91b, CL91a]. *B* provides a platform for solving the problem specification and correct construction of software systems. It is a flexible inference engine which forms the basis of a computer-aided system for the formal construction of provably correct software. Using a mechanized theorem prover to validate a proof presents an additional burden for the user, since machine validated proofs are longer and more difficult to produce. However, if one trusts the theorem prover, one may then focus attention on the specification that was proved. This analysis may be facilitated by consulting the mechanized proof script.

The design of the programming environment consists in several steps that are either automatic, or semi-automatic (Figure 1). The first step consists in writing a METAL specification of the *Unity* language. This specification defines the concrete syntax, the abstract syntax and the rules of trees formation that express the correspondence between abstract and concrete syntax. The METAL-PPML generates tables and programs used to generate a parser from this specification. The generation of a parser is not completely automatic and the user has to supply some files names along with those generated by METAL-PPML. The semantics of the language is handled by the TYPOL environment. The second step writes the PPML specification of correctness the rules of textual representation (or unparsing) for the *Unity* formalism from its abstract syntax. The unparser for the *Unity* formalism is generated using the *compile* command of the METAL-PPML

*on sabbatical leave at the department of Computing Science University of Stirling under the European Science Exchange Programme Royal Society - CNRS

[1] This division is taken from Jones (Systematic Software Development Using VDM, 1990)

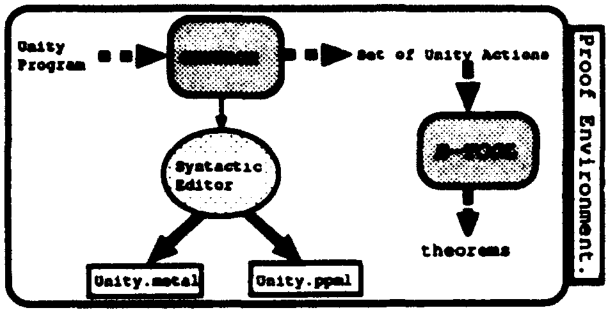


Figure 1: The Proof Environment

environment. The *Unity* environment comprises two kinds of editors: textual and structural. The user can easily write a *Unity* program in a textual form. A parser checks it. If the program is syntactically correct, the parser generates the internal representation. The user can run an interface to the theorem prover that allows him to prove the correctness of the *Unity* program using the set of its actions (statements). The interface ensures the interaction between the *Unity* environment and the proof system implemented under *B*. The interface operates on the internal representation.

The prover is designed according to the *enrichment principle*. A basic layer represents the Dijkstra's *wp*-calculus [Dij76]. This is successively enriched with other theories for reasoning on *Unity* programs. To *wp-theory*, we have supplied another layer for deriving *safety* properties which we denote by *unless-thy*. *Ensures-thy* and *leads-to-thy* define the most interesting progress properties (Figure 2).

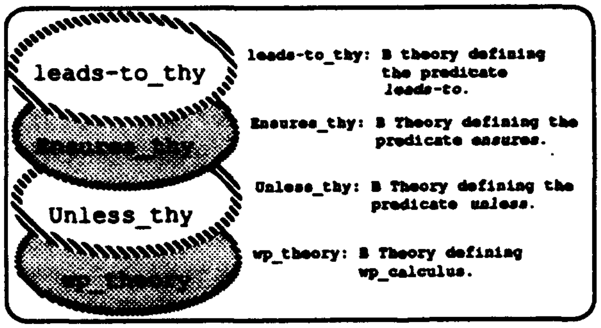

leads-to_thy
Unless_thy
leads-to_thy: B theory defining the predicate leads-to.
Ensures_thy: B Theory defining the predicate ensures.
Unless_thy: B Theory defining the predicate unless.
wp_theory: B Theory defining wp_calculus.

Figure 2: Structure of the Proof System

Keywords: Automated theorem proving, concurrency, program verification, formal specifications, Unity, B-Tool.

References

[BM93] N. Brown and D. Méry. A Proof Environment for Concurrent Programs . In *Proceedings FME93 Symposium*. Springer Verlag, 1993. To appear.

[BS91] R.J.R. Back and K. Sere. Deriving an Occam Implementation of Action Systems. In C. Morgan and J.C.P. Woodcock, editors, *4rd Refinement Workshop*. Springer-Verlag, January 1991. BCS-FACS, Workshops in Computing.

[CL91a] BP Innovation Centre and Edinburgh Portable Compilers Ltd. *B-Tool Version1.1, Reference Manual*, 1991.

[CL91b] BP Innovation Centre and Edinburgh Portable Compilers Ltd. *B-Tool Version1.1, Tutorial*, 1991.

[CL91c] BP Innovation Centre and Edinburgh Portable Compilers Ltd. *B-Tool Version1.1, User Manual*, 1991.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Kna90] E. Knapp. An Exercise in the Formal Derivation of Parallel Programs: Maximum Flows in Graphs. *Transactions On Programming Languages and Systems*, 12(2):203–223, 1990.

[M92́] D. Méry. The \⊓ System as a Development System for Concurrent Programs: δ\⊓. *tcs*, 94(2):311 - 334, march 1992.

The ASF+SDF Meta-environment

A. van Deursen, T.B. Dinesh, and E A. van der Meulen

CWI - P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.
Email: {*arie,dinesh,emma*}*@cwi.nl*

System Demonstration AMAST'93

Introduction In addition to facilitating formal reasoning about software, algebraic specifications provide means for rapid prototyping [1]. In particular, this can be applied to specifications of various aspects of programming languages, thus obtaining tools that can be part of a programming environment for the language specified. In Amsterdam, at CWI and UvA, the GIPE[1] group has been studying these topics. Thus far, this has resulted in:

- An algebraic specification formalism, ASF+SDF, especially designed for defining the syntax and semantics of programming languages [1, 4];
- The ASF+SDF tool generator, deriving parsers and term rewriting machines from algebraic specifications [5];
- The ASF+SDF Meta-environment, giving support when developing ASF+SDF specifications [5]

The ASF+SDF formalism and system are especially designed to support easy specification of all relevant properties of programming languages: syntax, static semantics, dynamic semantics, transformations, and so on.

The ASF+SDF Formalism The ASF+SDF formalism is the result of the "marriage" of ASF [1] with SDF [4]. ASF is an Algebraic Specification Formalism, supporting many-sorted first-order signatures, (conditional) equations, and modularization. SDF is a Syntax Definition Formalism, defining lexical, concrete, and abstract syntax all at once. Each SDF rule corresponds both to a context-free grammar production, and a function declaration in a signature.

The ASF+SDF System From an SDF definition, a parser can be derived, which in turn can be used to derive a syntax-directed editor. The equations of an ASF+SDF module can be executed as term rewriting systems. Both the parsers and the term rewriting systems are generated incrementally, so small updates in the specifications lead to adaptations rather than regenerations from scratch.

[1] Partial support has been received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II) and from the Netherlands Organization for Scientific Research - NWO, project *Incremental Program Generators*

The ASF+SDF system and formalism have been used succesfully for the derivation of environments for (subsets of) λ-calculus,Eiffel, Action Semantics, modelling of financial products, Pascal, Lotos and so on.

Current Research Current research activities include incremental rewriting (small changes in the initial term cause adaptations of the normal form rather than recomputation from scratch) [7]; origin tracking (automatically maintaining relations between initial term and normal form, with applications to the generation of error handlers and run-time animators from specifications of static or dynamic semantics of programming languages) [2]; generation of C-code from algebraic specifications; customizable user-interface for generated environments [6]; and experiments with the use of an abstract-interpretation style for specifictaion and generation of type checkers [3].

More Information More information on the ASF+SDF system can be obtained by anonymous ftp: get file abstracts.ps.Z from ftp.cwi.nl in directory pub/gipe.

References

[1] BERGSTRA, J., HEERING, J., AND KLINT, P., Eds. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[2] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. Tech. Rep. CS-R9230, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992. To appear in the *Journal of Symbolic Computation*, special issue on Automatic Programming, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe.

[3] DINESH, T. Type checking revisited: Modular error handling. Tech. Rep. CS-R93xx, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1993. To appear.

[4] HEERING, J., HENDRIKS, P., KLINT, P., AND REKERS, J. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices 24*, 11 (1989), 43–75.

[5] KLINT, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology 2*, 2 (1993). To appear. Preliminary version in J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, LNCS 490, 1991.

[6] KOORN, J. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992.

[7] MEULEN, E. V. D. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990. Extended abstract in M. Nivat, C. Rattray, T. Rus and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'91)*, Workshops in Computing, Springer-Verlag, London (1992) 277–286.

Executing Action Semantic Descriptions using ASF+SDF

Arie van Deursen
CWI - P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: *arie@cwi.nl*

Peter D. Mosses
Computer Science Department, Aarhus University DK-8000 Aarhus C, Denmark
Email: *pdmosses@daimi.aau.dk*

System Demonstration AMAST'93

Introduction *Action Semantics* is a framework for describing the semantics of programming languages [Mos92]. It is based on:

- *Action Notation*, used for expressing so-called *actions*, which represent the semantics of programming constructs; and
- *Unified Algebras*, used for specifying the data processed by actions, as well as for defining the abstract syntax and semantic functions for particular programming languages, and the symbols used in Action Notation.

Currently, only little tool support for action semantics exists. Tool support, however, becomes more and more important, now that an increasing number of researchers and practitioners start using action semantics. Having simple tools that perform parsing, editing, checking or interpretation of action semantic descriptions is essential when writing large specifications.

In order to obtain these tools, the ASF+SDF[1] [BHK89, Kli93] approach to tool generation from algebraic specifications of programming languages came to mind. The syntax of a language is described using the Syntax Definition Formalism SDF, which defines context-free syntax and signature at the same time. Functions operating on terms over such a signature are defined using (conditional) equations. Typical functions describe type checking, interpreting, compiling, etc. of programs. These functions are executed by interpreting the algebraic specifications as term rewriting systems. Moreover, from SDF definitions parsers can be generated, which in turn are used for the generation of syntax-directed editors[2].

The MetaNotation Unified Algebra definitions are written in the *MetaNotation*. A syntax of the MetaNotation has been given in [Mos92, Appendix F], which we have transformed into an SDF definition. Although the MetaNotation supports a great deal of syntactic freedom, a context-free grammar could be given by choosing a liberal syntax for symbols and terms. This automatically resulted in a generated syntax-directed editor for the MetaNotation.

[1] ASF+SDF is an abbreviation for Algebraic Specification Formalism + Syntax Definition Formalism

[2] During AMAST'93, a separate demonstration of ASF+SDF is given as well [DDM].

Checking MetaNotation Modules In the MetaNotation, symbols can be introduced and given functionalities, and then be used in formulae (equations). With the ASF+SDF parser generator at hand, an easy way to check consistency between definition and use, is to derive SDF rules from functionality declarations, and to use these rules to try to parse the formulae. Thus we have written, in ASF+SDF, a translator taking a MetaNotation module as input and producing SDF rules from each functionality declaration in that module.

Executing MetaNotation Modules Though the formulae allowed in the MetaNotation can be very general, a substantial number of equations in it (in particular, the equations defining semantic functions) can be interpreted as rewrite rules. Thus, we have written a translation function in the ASF+SDF formalism, taking a MetaNotation module as input and producing ASF equations.

Tool Summary In summary, we have given algebraic specifications of (1) the abstract syntax of the MetaNotation, (2) a function translating MetaNotation function declarations to many-sorted signatures, and (3) a function mapping MetaNotation equations to rewrite rules. Using the ASF+SDF Meta-environment to execute these specifications has resulted in the following tools:

- Parsing and syntax-directed editing of MetaNotation descriptions;
- Checks on use of sorts for functions introduced in MetaNotation descriptions;
- Translation of MetaNotation modules to corresponding ASF+SDF modules, allowing, e.g., execution of MetaNotation descriptions as term rewriting systems, as well as generation of parsers from grammar definitions given in MetaNotation.

In the demonstration, we will illustrate the use of these tools by showing the action semantic description of a small imperative language called *Pico*. We will see syntax-directed editing of this definition, incremental generation of ASF+SDF modules from it, syntax-directed editing of Pico programs based on the generated SDF definition, and translation of Pico programs to ActionNotation by interpreting the semantic equations as rewrite rules.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[DDM] A. van Deursen, T.B. Dinesh, and E.A. van der Meulen. The ASF+SDF meta-environment. System Demonstration AMAST'93.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology,* 2(2), 1993.

[Mos92] P.D. Mosses. *Action Semantics,* volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

The LOTOS Toolbox

Thony van der Vloedt
Information Technology Architecture B.V.
Institutenweg 1
7521 PH Enschede
The Netherlands
Phone: +31 53 309682
Fax: +31 53 309669
email: vdvloedt@ita.nl

The LOTOS toolbox is a coherent set of tools in support of the ISO standard (8807) Formal Description Technique LOTOS. This language is theoretically based on process algebra. For data typing the Abstract Data Type language ACT-ONE is used. LOTOS finds it main application in the area of distributed systems and data communications.

One of the initial goals of the language was to be able to specify in a precise, yet implementation free way, the OSI data communication standard services and protocols. Currently, for many OSI standards related Working Papers exist in which the protocol or service is formally specified in LOTOS.

LOTOS can also be utilized to aid in the design of distributed systems. The advantages of usage of LOTOS in design include increased precision in the communication between designers mutually, and between designers and future users of the system, improved quality of the system through tool supported validation and testing, and animation and prototyping allowing early assessment of the system to be built.

Tool overview

The LOTOS toolbox contains a number of cooperating tools supporting the specification and implementation of LOTOS specifications. The toolset includes the following tools:

- the TOPO front-end syntax checking and static semantic checking,
  This tools produces a LOTOS specification in Common Representation (CR) format which is used as input by other tools,
- the structure editor CRIE
  The structure editor guides the user in the correct use of LOTOS and provides syntax and static semantic checking on the fly. It also produces CR format specifications.
- the system validator SMILE,
  provides symbolic execution of LOTOS. SMILE allows the user to dynamically analyse the behaviour of his specification (CR format) by stepping through allowable events,
- the graphical browser GLOW,

transforms a textual LOTOS specification (CR format) in a graphical representation according to the graphical LOTOS standard,

- the TOPO back-end C-code generator,
  "compiles" an implementation oriented LOTOS specification into a prototype which can be used for early evaluation of the designed system

Available platforms:

Sun 3,Sun 4, SunOS 16Mb memory, 35 Mb disk
HP, HP Unix, 16Mb memory, 35Mb disk

The tools are commercially available.